

LIBRARY AND DATA BASE TECHNOLOGY

BS(LIS)

Code No. 9213

Units: 1-9



Department of Library and Information Sciences
Faculty of Social Sciences and Humanities
ALLAMA IQBAL OPEN UNIVERSITY
ISLAMABAD

LIBRARY AND DATABASE TECHNOLOGY

BS (LIS)

Course Code: 9213

Units: 1–9

AIOU website: <https://aiou.edu.pk>
LIS Department website: <https://lis.aiou.edu.pk/>
LIS Facebook page: LIS@AIOU official



ALLAMA IQBAL OPEN UNIVERSITY
Department of Library and Information Sciences

(All Rights Reserved with the Publisher)

Year of Publication 2022

Quantity 1000

Layout Setting Muhammad Zia Ullah

Incharge Printing Dr. Sarmad Iqbal

Printer..... Allama Iqbal Open University

Publisher Allama Iqbal Open University,
Islamabad

COURSE TEAM

Chairman Course team:

Dr Pervaiz Ahmad
Professor/Chairman

Course Development Coordinator:

Dr Amjid Khan
Assistant Professor

Reviewed by

Dr Pervaiz Ahmad
Muhammad Javad
Dr Muhammad Arif

CONTENTS

	<i>Page #</i>
FOREWORD	v
PREFACE.....	vi
ACKNOWLEDGEMENTS	vii
INTRODUCTION	viii
ASSESSMENT/EVALUATION OF STUDENTS' COURSEWORK	ix
OBJECTIVES	ix
COMPULSORY READING.....	x
ADDITIONAL READINGS	x
UNIT NO. 1 INTRODUCAION AND DATABASE BASICS	7
UNIT NO.2 SETUP AND ADMINISTRATION	23
UNIT NO. 3 INTRODUCTORY PROGRAMMING	33
UNIT NO. 4 CREATING REPORTS	47
UNIT NO. 5 PROJECT DESIGN.....	65
UNIT NO. 6 PROGRAMMING THE APPLICATION.....	85
UNIT NO. 7 SECURTIY-RELATED TECHNIQUES.....	117
UNIT NO. 8 CREATING PUBLIC INTERFACES	129
UNIT NO. 9 DEVELOPMENT PROCEDURE	147

FOREWORD

Department of Library and Information Sciences was established in 1985 under the flagship of the Faculty of Social Sciences and Humanities intending to produce trained professional manpower. The department is currently offering seven programs from certificate courses to PhD levels for fresh and/or continuing students. The department is supporting the mission of AIOU keeping in view the philosophies of distance and online education. The primary focus of its programs is to provide a quality education by targeting the educational needs of the masses at their doorstep across the country.

BS 4-year in Library and Information Sciences (LIS) is a competency-based learning program. The primary aim of this program is to produce knowledgeable and ICT-based skilled professionals. The scheme of study for this program is specially designed on the foundational and advanced courses to provide in-depth knowledge and understanding of the areas of specialization in librarianship. It also focuses on general subjects and theories, principles, and methodologies of related LIS and relevant domains.

This new program has a well-defined level of LIS knowledge and includes courses in general education. The students are expected to advance beyond their higher secondary level and mature and deepen their competencies in communication, mathematics, languages, ICT, general science, and an array of topics of social science through analytical and intellectual scholarship. Moreover, the salient features of this program include practice-based learning to provide students with a platform of practical knowledge of the environment and context, they will face in their professional life.

This program intends to enhance students' abilities in planning and controlling library functions. The program will also produce highly skilled professional human resources to serve libraries, resource access centres, documentation centres, archives, museums, information centers, and LIS schools. Further, it will also help students to improve their knowledge and skills of management, research, technology, advocacy, problem-solving, and decision-making relevant to information work in a rapidly changing environment along with integrity and social responsibility. I welcome you all and wish you good luck with your academic exploration at AIOU!

Prof. Dr. Zia Ul-Qayyum
Vice-Chancellor

PREFACE

In a digital environment, the relational database is the most efficient way to manage large lists of information, limiting the number of places you need to store data to one place. With their efficiencies in maintaining data, relational databases are well suited to the highly structured data of libraries. Similarly, in today's globally networked computer environment, people increasingly expect to interact with a library's collections and services through a web browser. This means that libraries must maintain a web presence, if not web servers. Creating and maintaining small sets of static HTML files (perhaps a hundred) is not too difficult. On the other hand, when you start maintaining sets of thousands of pages, the process gets old quickly. Moreover, the implementation of relational databases forces you to think very critically about the data being stored and reported upon. Thinking critically about the data, information, and knowledge is a core characteristic of librarianship. It seems as if relational databases were made expressly for libraries and librarians.

This course book provides solutions to the challenges of maintaining lists of data in an era of globally networked computers. *First*, it describes in detail what it means to design and maintain a relational database. *Second*, it demonstrates how to write computer programs in the open-source language PHP, against the relational database to generate HTML pages and search the database's content. By exploiting these two techniques it is possible for a minimum of people, with a diverse set of skills, to maintain large sets of data and distribute that data immediately and accurately on the web. If you take advantage of the techniques described in this study guide, you will likely be able to drastically reduce the amount of time you spend writing HTML. You and your library will be able to work more efficiently. Content specialists will be able to focus more on content, and infrastructure providers will be able to focus on access. This study guide empowers you to put into practice the principles of librarianship in a "webbed" environment. The students will learn two things from this study guide. *First*, the students will learn how to collect, organize, and disseminate data, information, and knowledge in a digital, globally networked environment. *Second*, they will learn how to do this in an open-source environment. Furthermore, this course is designed for library and information sciences students with the purpose to prepare them for their future roles in an electronic environment. The expected learning outcomes of this course include a combination of knowledge and skills with an emphasis on its use in professional endeavors.

Prof. Dr Syed Hassan Raza
Dean, Faculty of Social
Sciences & Humanities

ACKNOWLEDGEMENTS

First, I am extremely grateful to the worthy Vice-Chancellor and the worthy Dean, FSSH for allowing me to prepare this book. Without their support, this task may not be possible. Further, they have consistently been a source of knowledge, encouragement, benignancy, and much more.

I am highly indebted to my parents, spouse, siblings, and children, who allowed me to utilize family time in the completion of this work timely. Their continuous prayers kept me consistent throughout this journey. I would also appreciate the cooperation of my departmental colleagues extended to me whenever required. Special thanks to the Academic Planning and Course Production (APCP) and Editing Cell of AIOU for their valued input that paved my path to improve and finish this book following AIOU standards and guidelines. They have been very kind and supportive as well.

I would also like to thank the Print Production Unit (PPU) of AIOU for their support regarding the comprehensive formatting of the manuscript and for designing an impressive cover and title page. Special thanks are also owed to AIOU's library for giving me the relevant resources to complete this task in a befitting manner. I am also thankful to ICT officials for uploading this book to the AIOU website. There are many other persons, whose names I could not mention here, but they have been a source of motivation in the whole extent of this pursuit.

Dr Amjid Khan
Assistant Professor, LIS

INTRODUCTION

This course has been organized in a way to help you in completing your required coursework. There are nine units in this course. Each unit starts with an introduction, which provides an overall overview of that unit. The introduction part is followed by objectives in each unit that shows the basic learning purposes. Similarly, the rationale behind these objectives is that after reading the unit a student should be able to explain, discuss, compare, and analyze the concepts studied in that unit. Hence, this study guide is intended to be a concise appetizer and learning tool in which the course contents are briefly introduced.

This study guide is based on prescribed reading materials. For each unit, these prescribed reading materials have been classified as compulsory readings and suggested readings. Students are bound for studying these materials to have successful completion of the course. After every unit, self-assessment questions and activities have been put forth for the students. These questions are meant to facilitate students/you in understanding how much student/you have learned.

For this course, workshop and tutorial support will be provided as per AIOU policy. So, before going to attend a class, prepare yourself to discuss course contents with your tutor. There will be 70% compulsory attendance in every workshop. After completing the study of the first 5 units 'Assignment No. 1' is due. The second assignment that is 'Assignment No. 2' is due after the completion of the next 4 units. These two assignments are to be assessed by the relevant tutor/resource person. Students should be very careful while preparing the assignments because these may also be checked with Turnitin for plagiarism.

COURSE STUDY PLAN

As you know the course is offered through distance education, so it is organized in a manner to evolve a self-learning process in absence of formal classroom teaching. Although the students can choose their way of studying the required reading material, but advised to follow the following steps:

- Step 1:** Thoroughly read the description of the course for clear identification of the reading material.
- Step 2:** Carefully read the way the reading material is to be used.
- Step 3:** Complete the first quick reading of your required study materials.
- Step 4:** Carefully make the second reading and note down some of the points in your notebook, which are not clear and need full understanding.
- Step 5:** Carry out the self-assessment questions with the help of study material and tutor guidance.
- Step 6:** Revise notes. It is quite possible that many of those points, which are not clear and understandable, previously become clearer during the process of carrying out self-assessment questions.
- Step 7:** Make a third and final reading of the study material. At this stage, it is advised to keep in view the homework (assignments). These are compulsory for the successful completion of the course.

ASSESSMENT/EVALUATION OF STUDENTS' COURSEWORK

Multiple criteria have been adopted to assess students' work for each course, except for Research Project/Project as under:

- (a). Written examination to be assessed by the Examination Department, AIOU at the end of each semester = 70% marks (pass marks 50%). AIOU examination rules apply in this regard.
- (b). Two assignments and/or equivalent to be assessed by the relevant tutor/resource person = 30% marks (pass marks 50% collectively).

All the matters relating to Research Project/Project will be dealt with as per AIOU rules. However, the pass marks for Research Thesis are 50% both in the evaluation of the research report and viva voce examination separately.

OBJECTIVES

After studying this course, you will be able to learn:

- 1. Database basics, setup, administration, and programming
- 2. Creating reports
- 3. Project design
- 4. Programming the application
- 5. Security-related techniques
- 6. Creating public interfaces
- 7. Development procedures

COMPULSORY READING

Westman, S. R. (2006). *Creating database-backed library web pages: Using open source tools*. Chicago: American Library Association. Available at:
[https://pdf-drive.com/download.php?title=Creating%20Database-Backed%20Library%20Web%20Pages:%20Using%20Open%20Source%20Tools%20\[1ed.\]0838909108,%209780838909102,%209780838998489&content=&downlurl=](https://pdf-drive.com/download.php?title=Creating%20Database-Backed%20Library%20Web%20Pages:%20Using%20Open%20Source%20Tools%20[1ed.]0838909108,%209780838909102,%209780838998489&content=&downlurl=)

ADDITIONAL READINGS

Baruah, A. (2002). *Library database management*. Gyan Publishing House. Chicago: American Library Association. Available at:
<https://books.google.com.pk/?hl=en>

Krier, L., & Strasser, C. A. (2014). *Data management for libraries: LITA guide*.

Preston, C., & Lin, B. (2002). Database technology in digital libraries. *Information Services & Use*, 22(1), 9-17.

Singh, P. (2004). Library databases: Development and management. *Annals of Library and Information Studies*, 51(I), 72-81. Available at:
<http://nopr.niscair.res.in/bitstream/123456789/7488/1/ALIS%2051%282%29%2072-81.pdf>

Suseela, V. J., & Uma, V. (2017). *Data management for libraries: Understanding DBMS, RDBMS, IR technologies & tools*. Ess Ess Publication

UNIT-1

INTRODUCTION AND DATABASE BASICS

Compiled by: **Dr Amjid Khan**

Reviewed by: **1. Dr Pervaiz Ahmad**
2. Muhammad Jawwad
3. Dr Muhammad Arif

CONTENTS

Page #

Introduction	9
Objectives	9
1.1 What Is Database-Backed Web Pages?	10
The Old-Fashioned Approach.....	10
Creating Static Webpage Reports.....	10
Creating Dynamic Web Page Reports	10
1.2 What Can Database-Backed Pages Be Used For?	11
1.3 Open-Source Software	11
1.4 Database Basics	12
Data Management Approaches.....	12
Structured Text Files.....	12
Comma-Separated Values	13
Single-Table, Fixed-Field.....	13
Personal Bibliography.....	13
Personal Information Managers (Pims).....	14
Relational Databases (Rdbms).....	14
Object-Oriented Databases.....	14
XML Databases	14
Marc Systems	15
1.5 Relational Database Management Systems	16
1.6 Tables And Fields.....	18
1.7 Designing Good Databases	19
1.8 Structured Query Language (SQL).....	19
Database Maintenance	19
Data Security	19
Database Integrity.....	20
Table/Row Locking.....	20
Other Features	20
1.9 Stored Procedures And Triggers	20
1.10 Choosing An Rdbms	21
1.11 Self-Assessment Questions	21
1.12 Activities.....	21
1.13 References	22

INTRODUCTION

A database-driven website uses a database for collecting and storing information. A database stores information in a structured way using tables on a web server. Web pages are created on the fly using programming languages (such as PHP), which store and retrieve data in the database. This unit covers an overview of the ordinary database structure, creating static and dynamic web reports, database-backed webpages, open-source software, and the basic structure of the database. It also describes the Personal Information Managers (PIMs), Relational Databases (RDBMS), XML Databases, MARC Systems, Relational Database Management Systems and Structured Query Language (SQL).

OBJECTIVES

After reading this unit you will be able to

- Database-backed webpages
- Open-source software
- Database basics and Database Structure
- Personal Information Managers (PIMs)
- Relational Databases (RDBMS)
- XML Databases
- MARC Systems
- Relational Database Management Systems
- Structured Query Language (SQL)

1.1 WHAT ARE DATABASE-BACKED WEB PAGES?

The first thing to explain about database-backed Web pages is what they are. Right now, you're possibly asking yourself: "Don't you write HTML pages using Web editors, such as Dreamweaver or FrontPage, and then transfer them to a Web server? What do databases have to do with publishing Web pages?" The answer is that, although they differ in detail, all database-backed Webpages are essentially Web-based reports. Many of you are already familiar with the reports (such as overdue notices) that your local online library system produces. You (or those who create your reports) write the programs that tell the system what pieces of information you want. The programs then respond to user input and:

- search the database using the query designed into the report
- compile a list of information that matches the designated criteria
- take the data and format them using rules built into the program
- print the report

Web pages are no different in their basics in that they also search a database, embed the results

within HTML tags and output the results to a Web server.

The Old-Fashioned Approach

To begin understanding how database-backed Web pages work, let's look at how a typical Web page is retrieved. The publisher (Web page creator), usually using Web-editing software of some kind, creates a file containing the data the user is intended to see, formats it using HTML, and places it on the Web server. A user who wants to view the information then clicks on a link to request the page, which sends a request (via a URL) for that particular page to the Web server, and the Web server retrieves that file and sends it to the browser for viewing. Note that creating and retrieving are separate processes: the only time the information that the user sees changes is when the publisher edits the page and uploads it to the server.

Creating Static Webpage Reports

Database-driven information can be created in two ways. In the first, the publisher enters a query using Web report writing software and the query is then sent to the database. The database is searched, and the results are returned to the report writer, which then formats the results as a Web page. The publisher then transfers the file to the Web server, where the user retrieves it in the same manner.

Creating Dynamic Web Page Reports

The process of creating a dynamic database-backed Web page is like static Web page reports, with one important difference. Once the programmer has written the program to search the database and output the data as an HTML report, the only person involved in the Web page creation is the user, who initiates the process either by clicking on a link or by filling in a search form and clicking the **Submit** button. When this happens, the browser sends two types of information to the server: the name and location of the program that will create the report; and the parameters for that report. The server then notes that the

requested resource is a program, and it then calls the requested program and passes the appropriate parameters to it.

The program takes the parameters and constructs one or more queries, opens a connection to the database, and passes the query or queries to it. The database then executes the requested search or searches and creates a retrieval set, which it then passes back to the program, where the results are formatted as an HTML document. The resulting Web page is then returned to the Web server, which treats it as a Web page and passes it back to the browser, where it is displayed. For this process, you need a database, a program or application, and a Web server.

1.2 WHAT CAN DATABASE-BACKED PAGES BE USED FOR?

It is difficult to provide hard and fast rules, certain guidelines may help.

In general, databases can be useful where:

- Data are made up of discrete pieces of information about persons or things.
- Data are highly and regularly structured.
- Information changes often or reuse of the same data in a variety of contexts is desirable.
- You wish to search for discrete elements within the data.

On the other hand, databases—particularly the type with which we will be dealing—are not well suited for pages that:

- Are very complex and have significant amounts of textual data
- Contain hierarchically organized data.
- Are more document-like and requires a specialized layout.

1.3 OPEN-SOURCE SOFTWARE

The open-source movement is one of the most exciting developments in software in years. Using a model of cooperation and sharing, rather than competition and profit, open-source licensing allows users to freely (most of the time) download and modify and redistribute the software without incurring licensing costs. The advantages of such an approach are many and powerful:

- Open source provides an extremely low-cost alternative to commercial systems.
- Open-source tools are usually -cross-platform and can run on all forms of Unix (including Linux), various flavors of Windows, and increasingly on Macintosh OS X.
- Open-source products are extremely powerful and provide most—if not all (and in some cases more)—functionality that one can find in commercial products.
- Product development is undertaken by individual developers rather than a central commercial entity.
- Open-source programs are extremely popular.

Another reason the open-source movement is valuable to librarians is the number of similarities between it and librarianship:

- Both seek to make their product, be they program or information, as freely available as possible.
- Both are based on a collaborative philosophy, not dissimilar to peer review, in which cooperation rather than profit-seeking and a group approach to developing the product and solving problems are fundamental tenets.
- Both seek to disseminate information in such a way that it furthers the quality and diversity of the products and services they provide.
- Both seek to empower the user rather than maintain centralized control of development and distribution.

1.4 DATABASE BASICS

Here we will look at some essentials of database management systems. We begin by exploring various approaches to storing and retrieving data, focusing in particular on relational database management systems (RDBMS). We then examine elements of database structures and modelling and see how we can use them to design solid databases for use in our Web applications. After that, we introduce you to Structured Query Language (SQL) and show how we can use it to interact with databases. We then present some of the techniques RDBMSs provide for ensuring data integrity and security. We finish up by discussing various open-source database implementations currently available.

Data Management Approaches

Although data management techniques may come in many shapes, sizes, and colours, they all have one basic characteristic in common: they structure the information that they store. Putting each piece of data in its own unique and identifiable cubbyhole allows programs to find and display it when requested. The following two approaches are used to structuring data: structured text files and database management software.

Structured Text Files

A delimited text file is an ASCII file in which we store data, placing a delimiter (specific character) between each piece of information (field) that we want to store. This effectively tells a program where one field ends and the next one begins. A delimited text file is similar to a spreadsheet, in which each row (or record) represents a single thing (book, or person, for example) and each row has several fields (title, author, and the like) separated from each other by the delimiter. You then write programs in languages such as Perl that will parse the data (break them down into their constituent components) and look through those components to find and output the information contained in these files.

Comma-Separated Values

Probably the most used delimited text file format is comma-separated values (CSV). It is used by programs such as Microsoft Access and Excel as one way of exporting and importing data. CSV files have quotation marks around each field and a comma separating

the fields. A CSV version of our example above would look like this: "Smith, John", "jsmith@mylib.net", "555-1212"
"Doe, July", "jdoe@mylib.net", "555-2121""Jones, Fred", "", "555-3123"

Database Structure

Database structure includes the following components:

- *Database*. The basic container in a database management system, a database contains the table or tables in which we place our data.
- *Table*. Like the delimited file, a table is the base unit within a database that contains the actual data. Tables generally store information on a specific and (if the database is well structured) unique entity (concept), such as books, authors, orders, and such.
- *Record*. The structure within tables that contains descriptions of individual instances of the entity that the table in question represents, records are the database equivalent of the rows in the delimited file.
- *Field*. Database fields are the individual parts of the record that describe specific aspects of the entity being represented in an individual record (such as the individual elements contained between the vertical bars above).
- *Index*. Just as books contain indexes to help you find information quickly, database systems create similar indexes that facilitate quick and easy retrieval of information from the database. Some types, such as UNIQUE indexes, can enforce certain rules about data entry.

Single-Table, Fixed-Field

Database programs that store all information in a single table were very popular in early PC databases and information managers. Packages such as PC-File and VP-Info provided a quick and inexpensive way of maintaining simple information collections, such as address lists and phone numbers. Unfortunately, several problems limited their usefulness:

- Data in one database created using these packages were inaccessible to any other database.
- Each record was required to contain all possible fields. In most cases, these were fixed lengths, which meant that they took up the same amount of space on the disk whether the field contained any data or not. Both features meant an often-inordinate amount of duplicated information, wasted space, and unused space.
- Customized data entry screens were generally not an option.

- Few if any of these programs are Web aware. To use the data in such systems, you need to extract their information and load it into a Web-aware database.

Personal Bibliography

Like single-table databases, what is known as personal bibliography software has been extremely popular in the library community for several years. These packages use variable length fields, which take only as much space as is necessary to store the data. They still have the drawback of using only a single table but have been invaluable in helping users create bibliographies, pathfinders, and a world of information support tools. Examples include Notebook II, ProCite, EndNote, and Library Master Balboa Software.

Personal Information Managers (PIMs)

A type of application very popular in the 1990s, PIMs (as they were called) allowed you to place all sorts of different kinds of unstructured data into the application, including numeric, date, and textual materials. They would then allow you to do a free text search through the database to find the information you needed.

Relational Databases (RDBMS)

RDBMS relational database management systems (RDBMS) were first developed in the 1970s. In its simplest form, this approach focuses on efficient storing, searching, and retrieving. Data are broken into separate concepts or components (tables) associated with a series of relationships.

Object-oriented databases

At the cutting edge of database technology, object-oriented database management systems (OODBMS) made quite a splash in some quarters. The idea is that data, rather than being stored in records, are stored as objects (individual data elements) that can be brought together on the fly by the calling program. They are becoming more readily usable, but are designed for large-scale applications, involve complex programming techniques and languages, and provide much more power and complexity than is required by the average library-created application.

XML Databases

Another new type of database product is the XML database. Designed to store and search/output XML documents using XPath or XQuery statements (or both), these programs can maintain the internal structure of XML documents and serve them up as needed. Their capacity to store and search hierarchically structured data makes them well adapted to dealing with such library-based resources as EAD finding aids and TEI-encoded documents. Although there are several open-source tools—such as eXist and Xindice—available, they address a different set of problems from those we deal with here and we will therefore not cover them.

MARC Systems

The five MARC 21 communication formats, MARC 21 Format for Bibliographic Data, MARC 21 Format for Authority Data, MARC 21 Format for Holdings Data, MARC

21 Format for Classification Data, and MARC 21 Format for Community Information, are widely used standards for the representation and exchange of bibliographic, authority, holdings, classification, and community information data in machine-readable form. A MARC record is composed of three elements: the record structure, the content designation, and the data content of the record. The **record structure** is an implementation of the international standard *Format for Information Exchange* (ISO 2709) and its American counterpart, *Bibliographic Information Interchange* (ANSI/NISO Z39.2). The **content designation**--the codes and conventions established explicitly to identify and further characterize the data elements within a record and to support the manipulation of that data--is defined by each of the MARC formats. The **content** of the data elements that comprise a MARC record is usually defined by standards outside the formats. Examples are the *International Standard Bibliographic Description* (ISBD), *Anglo-American Cataloguing Rules*, *Library of Congress Subject Headings* (LCSH), or other cataloguing rules, subject thesauri, and classification schedules used by the organization that creates a record. The content of certain coded data elements is defined in the MARC formats (e.g., the Leader, field 007, field 008).

The *MARC 21 Format for Bibliographic Data: Including Guidelines for Content Designation* defines the codes and conventions (tags, indicators, subfield codes, and coded values that identify the data elements in MARC bibliographic records. This document is intended for the use of personnel involved in the creation and maintenance of bibliographic records, as well as those involved in the design and maintenance of systems for communication and processing of bibliographic records. This documentation is also available online, including a concise version and a simple field list at: www.loc.gov/marc/.

Some earlier library systems did store their records in something approaching MARC format, but later systems usually do not. Although MARC is used to transfer records into and out of these systems, the way that the data are stored internally is quite different from the single-record MARC screen we have come to know and love. A growing number of systems are using relational database management systems—usually Oracle or Sybase—as their underlying database technology to store the data. Then, when a particular record is requested, these systems use programs to retrieve the data elements from wherever they reside and reconstruct a single MARC record on the fly for a display to the user. There may be times when the metaphor of the MARC record—which on the surface runs counter to many of the foundation principles of relational database design—may cause some confusion. Examples include the concepts of repeatable fields and subfields, and the idea of storing all data in a single record. In such cases, it may be helpful to remember the distinction between how the information is presented (as a MARC record) on the screen and how it is stored in the system.

1.5 RELATIONAL DATABASE MANAGEMENT SYSTEMS

To show how RDBMSs structure data, let's look at the differences between an RDBMS product (which is relational) and a single table database (which is not). We begin by examining how we might use a single-table database to create a checkout system for books (see figure 1.1)

title
publisher
copyright date
call number
bar code number
due date
patron name
patron address
patron city
patron state
patron zip code
patron phone number
patron email address

Figure 1.1

patrons				books			
Field	Data Type	Size	Key	Field	Data Type	Size	Key
patron_id	Character	40	P	title	Character	100	
first_name	Character	40		publisher	Character	100	
last_name	Character	40		patron_id	Character	12	F
address	Character	50		due_date	Date		
city	Character	50		call_number	Character	12	
state	Character	50		copyright_date	Date		
zip	Character	10					
email	Character	50					

Figure 1.2

Talk about wasted time, effort, and disk space (to say nothing of the possibility of errors)! Wouldn't it be a lot easier to put the patron information into one place and then point to it when needed? This is exactly what relational databases permit you to do. Rather than entering that information repeatedly, you merely create a second table—a patron's table—where information on the patron is kept. When a patron checks out a book, you place a pointer to the patron record inside the book's table. Then, by looking at the book record, you can follow the link back to the patrons' table to see who has checked out (see figure 1.2). Let's now look in some more detail at how relational database systems are put together.

Tables and Records and Links, Relational database management systems store information in multiple tables with each table representing a separate logical entity (concept) within the database. For example, in a library system, you can have a books table, a publishers table, a subjects table, and so on. For each separate entity within the database, a separate table will be created. There are two reasons why this is good.

First is that information on a given entity is stored in only one place. The second is that keeping entities separate from each other, allows the same information to be used in multiple ways. For example, a patron's list could be used to check out books, create a searchable phone directory, build a mailing list, or authenticate users for a library proxy server. Placing this information inside a book's record would make this reuse virtually impossible. Having broken down our information into separate tables, we need to find a way to bring them together as needed. We do this by creating links (relations) between the records that in turn link the various tables.

Tables are linked through what are called primary/foreign key pairs—a pair of fields, one in each table, that share a common value. We set relations—as in the example above—by taking a unique value (primary key) from a field in one entity and placing it in a corresponding field in another record in another table (foreign key) so that a link is established between the two records. For example, in figures 1-2 we place the `patron_id` (primary key) from the patrons' table into the `patron_id` (foreign key) field in the books table for the records that the user wishes to charge out. Then, when we later want to know what books a person has checked out, we look up the name, find out the person's patron id, and then search for all of the book's records that have that value in the patron id field in the books table.

As noted, a primary key is a unique value in a record (the value doesn't appear in that primary key field in any other record in that table) that is used to identify that—and only that—record. In our figure 1-2 example, the `patron_id` in the patrons' table is the primary key for that table (hence, the P in the **key** column). Tables need primary keys to identify an individual record in the database.

Foreign keys, on the other hand, are fields in one table into which the primary key of another table has been placed, thereby linking records in the two tables with this common piece of information. Because you may have multiple records in one table that point to a single primary key record in another table (patrons may have more than one book checked

out, for example), foreign keys are not required to be unique. Thus, in our example, the `patron_id` in the patrons' table (or `patrons.patron_id`) is the primary key and the `patron_id` field in the books table (or `books.patron_id`) contains the foreign key. One way to look at this is to think of the primary key as a surrogate (or, if you will, a database telephone number) for the record that contains it. As new linked records are added to the system, the database gives this phone number to the new record (placing it in the foreign key field) saying, "If you need this information, call this number."

In looking at figures 1-2, note that the arrow points from the books table to the patrons' table, even though we spoke about taking the `patron_id` from the latter and placing it in the former. This is because the function of the arrow is not to indicate the direction of movement of the linking information but to show where a field containing such information is pointing. In the next pages, we will be talking about some of the essentials of data modelling—the process of taking the pieces of information that you want to include in your system and structuring them in a way that will make storage and retrieval of that data more efficient.

1.6 TABLES AND FIELDS

In general, we will be using three types of tables in our applications in this study guide. Please note that these distinctions may sometimes blur in that a table may serve multiple purposes in a complex application:

- *Data tables* are the primary bearers of information within a database. They contain the information content that your application wishes to store and make available.
- *Linking tables* define many-to-many relationships, that is when we have multiple records in one table that are to be related to multiple records in another. We will discuss these in more detail.
- *Lookup (or authority) tables* restrict the set of values for a field to a predefined list stored in the table, such as the state abbreviations allowed by the U.S. postal service. There are generally three types of relationships that can exist between tables:
 - *One-to-one*. A record in one table is related to only one record in another table. This usually is done where information in the second table amplifies information contained in the first table or where, for security purposes, certain fields have been placed in a table with more restrictions on access. For example, in a personnel system, you may wish to keep public information on an employee (name, phone number, office number) separate from more sensitive information (SSN, salary, job performance rating, and the like).
 - *Many-to-many*. Multiple records in one table are related to multiple records in another table. Again using our online library system example, a single book table record may

be linked to multiple subject table records and a single subject table record may be linked to multiple book table records.

1.7 Designing Good Databases

In setting up databases, it is important to follow certain rules to ensure that finding information is both easy and predictable. This process—known as normalization—involves following standard procedures in designing and implementing the database. The following basic rules are recommended:

- Do not place more than one piece of information in a single field.
- Do not enter the same type of information (user’s name, subject heading, for example) into a given record more than once (for example, subject1, subject2, and so on).
- Do not enter the same information in more than one place. If you do, you will need to make changes in more than one table should the data change.
- Each table should be “about” only one thing. For example, don’t mix books and patron information on the same table.
- If the information could be used in more than one context, consider moving it to a separate table.

1.8 Structured Query Language (SQL)

Now that we have briefly looked at how a relational database structures data, let’s look at how we can interact with those structures. To support such interactions, a special language was developed concurrently with the relational database theory. Called Structured Query Language (or SQL), it is used to create queries (requests to the database to do something) that are used in all interactions with the system and partakes of many of the qualities of a language. SQL was intentionally designed to be understandable and constitutes a database lingua Franca that is used in virtually all RDBMSs. There may be individual differences in how each product implements its particular dialect of SQL, but the essentials of the language are the same. The structure portion of the acronym certainly lives up to its name. In their most basic form, SQL statements are built up from the following structured templates: *searching, adding, editing, and deleting*.

Database Maintenance

The three main SQL commands for updating the database include INSERT (for adding records), UPDATE (for updating records in the database), and DELETE (for deleting records). Let’s take a brief look at each of these in turn.

Data Security

One of the most important tasks any system has—particularly a Web-based system—is to protect the data from unauthorized access and manipulation. We need to make sure that only those people we want to have access to the data. To address this, all RDBMSs support user security.

Database Integrity

Beyond user security, we need to make sure that the integrity of the data is maintained. It is important not only that the information is inserted into the database correctly, but also that complete and proper relations are set and maintained. If these are not handled properly, severe problems can develop and make a system unusable.

Table/Row Locking

When working in a multiuser environment, one challenge is to ensure that two people do not try to save their edits on the same record at the same time. If that is permitted, the record can be corrupted, and the database compromised. To keep this from happening, database systems permit you to lock the resource you are editing, either at the table or the row level.

Two kinds of locks can be used. What is called a **WRITE** lock allows you to read/write data to the table/row and prevents everyone else from reading or writing until you finish your work and release the lock. What is known as a **READ** lock allows you and all others to read the table/row, but prevents everyone (including you) from writing to the table or row?

Other Features

Indexes: In creating a standard index, the database software creates a list of all the values contained in a field or fields and collects pointers to the records containing those values. Searching this index instead of the tables can dramatically increase the speed with which records are retrieved. RDBMS products can create other types of indexes as well. In the case of MySQL, there are three others, **PRIMARY**, **UNIQUE**, and **FULLTEXT**. On the one hand, indexing can make retrieval quicker when searching often-searched fields, particularly in large databases. On the other hand, overuse can slowdown record inserts/updates and takes up additional disk space.

Views: Views (sometimes called virtual tables) are a way of creating structures within the database across table lines. Such structuring involves taking fields from various tables and bringing them together and forming a logical construct (a sort of single virtual table) with which the programmer can interact.

1.9 Stored Procedures and Triggers

Stored procedures are pieces (modules) of programming code or SQL queries (or both) stored in the database for use by the database or applications. These modules have several advantages:

- They are precompiled and optimized so that they can run faster than ordinary routines.
- They are stored in the database and are available throughout the system.
- They reduce network traffic by allowing the module to be run within the server, rather than requiring that it be called from a network-based application.
- They are reusable and thus can reduce maintenance and recoding.

- They are inside the database and thus can execute code and access tables to which the user may not otherwise have access. This can be extremely valuable in that it can dramatically increase application security.
 - Triggers are essentially stored procedures that are automatically performed when changes are made to the table to which the trigger is attached. They can be extremely useful in maintaining data integrity and enforcing business rules.
- Support for both stored procedures and triggers is included in MySQL 5.0

1.10 Choosing an Rdbms

In selecting an open-source relational database management system, you have several choices, each with a varying degree of open sources and each with its pros and cons. The main candidates are:

- *MySQL*
- *PostgreSQL*.
- *Firebird*.
- *MSQL (or mini SQL)*.
- *OGenezzo*.

1.11 Self-Assessment Questions

- Q.1 What are database-backed web pages? Discuss with examples.
- Q.2 Define open-source software.
- Q.3 Explain database basics and database structure with examples.
- Q.4 Define personal information managers (PIMS).
- Q.5 Define relational databases management systems (RDBMS).
- Q.6 Define the following with examples.
- XML databases
 - MARC systems
 - Structured Query Language (SQL).
 - Database design

1.12 Activities

- Design a single-table database to create a checkout system for library books.
- Explore various approaches to storing and retrieving data, focusing on relational database management systems (RDBMS).
- Crete a sample of a 'Dynamic Web Page Report' of an academic library.

1.13 References

Baruah, A. (2002). *Library database management*. Gyan Publishing House.

[https://pdf-drive.com/download.php?title=Creating%20Database-Backed%20Library%20Web%20Pages:%20Using%20Open%20Source%20Tools%20\[1ed.%200838909108,%2009780838909102,%2009780838998489&content=&downlurl=](https://pdf-drive.com/download.php?title=Creating%20Database-Backed%20Library%20Web%20Pages:%20Using%20Open%20Source%20Tools%20[1ed.%200838909108,%2009780838909102,%2009780838998489&content=&downlurl=)

Chicago: American Library Association. Available at:

<https://books.google.com.pk/?hl=en>.

Krier, L., & Strasser, C. A. (2014). *Data management for libraries: LITA guide*.

Preston, C., & Lin, B. (2002). Database technology in digital libraries.

Information Services & Use, 22(1), 9-17.

Singh, P. (2004). Library databases: Development and management. *Annals of Library and Information Studies*, 51(I), 72-81.

<http://nopr.niscair.res.in/bitstream/123456789/7488/1/ALIS%2051%282%29%2072-81.pdf>

Suseela, V. J., & Uma, V. (2017). *Data management for libraries: Understanding DBMS, RDBMS, IR technologies & tools*. Ess Ess Publications

UNIT-2

SETUP AND ADMINISTRATION

Compiled by: **Dr Amjid Khan**

Reviewed by: **1. Dr Pervaiz Ahmad**
2. Muhammad Jawwad
3. Dr Muhammad Arif

CONTENTS

	<i>Page #</i>
Introduction	25
Objectives	25
2.1. Database Planning	26
2.2 Setting Up The Database	26
2.3 Defining The Data	26
2.4 Creating The User Account.....	28
2.5 Administration Tasks	28
2.6 Creating Indexes	29
2.7 Backups	29
2.8 Interaction Logging.....	30
2.9 Security	30
2.10 Self-Assessment Questions	31
2.11 Activities	31
2.12 References	32

INTRODUCTION

This unit describes some of the tasks involved in administering the database such as database planning, manual conversion, defining the data, creating the user account, creating indexes, backups, and security. At the end of the unit, self-assessment questions followed by practical activities are given to the students.

OBJECTIVES

After reading this unit you will be able to:

- Database planning
- Selecting appropriate fields
- Creating data structure?
- Establishing user accounts
- Administering the database,
- Database backup strategies and setting up user security and access controls.

2.1 DATABASE PLANNING

It is the management activities that permit the stages of the database system development life cycle to be realized as efficiently and effectively as possible. Database planning must be integrated with the overall IS strategy of the organization.

Before you create a database—to say nothing of the Web pages that the database will generate—you need to figure out how you want to use those pages, what data must populate the database for it to fulfil that purpose, and where to find that data. Say, for example, you want to create an online staff directory for your library. What information would you want to be included? Certainly, you would include the person's name, office address, and telephone number. You might also want to include the e-mail address, job title, type of position (civil service, faculty, administrative, and so forth), or areas of expertise.

In making this decision, it is helpful to define the purpose of your page: a directory, a contact list for departmental home pages, or a resource page to direct users to staff who can assist them. You might also consider how the database might be used in the future, such as an authentication mechanism for interactive Web applications. As you can see, a page's purpose strongly influences what data you need—and thus the database used to create the page.

2.2 SETTING UP THE DATABASE

Once you have decided what to include, you need to determine where to get the data. Additionally, you need to decide how you are going to get the data into your new system. You can take several different paths, but they boil down to two basic techniques. You can create the database in MySQL manually, obtain the data from the other system, and then load the data into the database you've just created. Alternatively, you can take the automated route, using products to connect to your current database and then loading the data directly into MySQL, creating your tables and loading the data in a single step.

Manual Conversion

A manual data load involves four steps: defining which data elements you wish to include; enveloping a way to download data from the existing system; setting up your MySQL database, defining the tables and fields to be included; and loading the data. We will take each of these steps in order.

2.3 DEFINING THE DATA

In our present staff directory example, we have decided what we want (personnel information with phone numbers) and where to get it (our library's HR system). Next, we sit down and identify

the fields with the data we want. Having decided that we want to create a simple address list, we look through the available pieces of information. We choose the following fields: last_name, first_name, phone, email, department, and location.

In preparing the list of data elements, you will need to know some things about each piece of data to be included in the database:

- *Name*. We all must have a name and fields are no exception.
- *Type*. What type of data is it? Is it a word or phrase? a number? a date?
- Data types most often encountered in relational databases include:
 - *char*—character data of a fixed length, causes every record in the table to occupy that length, whether it contains data or not, normally limited to 256 characters
 - *varchar*—character data in which there is likely to be wide variation in the length of the data going into that field, particularly useful in saving space, limited to 256 characters in MySQL
 - *text*—can store more than 256 characters of textual data
 - *integer*—numeric whole numbers
 - *float*—numbers with decimals
 - *date*—numeric, calendar dates that allow for doing calculations based on a date; normal format in MySQL is YYYY-MM-DD
 - *Y/N or Boolean*—one of two values—on and off—used to indicate whether something is or is not the case; when not available, as in MySQL, can be mimicked by a one-character Y/N *Field Size*.
- *Search*. Do you want to be able to search (or limit output) by this datum? If so, what type of searching? keyword? phrase? comparative (earlier than, less than, and so on)?
- *Description*. What is the usefulness of this field within the output? Why is it here? Where will the value for this field come from?

Table 2.1 provides an example of how we might fill in these values. It has seven data elements (six plus the primary key phone no—a necessary addition for several reasons). You also may note that we have put all the field names in lowercase. The reason for this is that MySQL, even on the Windows platform, does have times when it is case sensitive. Therefore, rather than spending time dealing with these vagaries, we have standardized on lowercase for all database, table, and field names, something that is normal in relational databases.

Table 2.1

	Name	Type	Size	Search	Description
1	Phone no	Auto	11	N	Primary key system generated
2	List-name	VC	50	Y	Person's last name-from field
3	First-Name	VC	50	Y	Person's first name-from frame field
4	Phone	VC	15	N	Phone number- from phone field
5	Email	VC	25	N	On-campus email address-from loc-email field
6	Department	VC	50	Y	Primary department-from dept field
7	Location	VC	60	Y	Office address-from address field

- **Conversion Programs**

Another approach to acquiring data is to use conversion programs to move the data directly into MySQL. This allows you to connect directly to the MySQL server (usually using an ODBC connection); automatically create databases, tables, and indexes; and then load the data into the resulting database. The advantage of this approach is that much of the work described is handled automatically (if the structure and data of the existing database will provide the output you desire).

2.4 CREATING THE USER ACCOUNT

One final thing needs to be done in MySQL to enable publishing data to the web. To run a report, you will need to create an account in the RDBMS that has permission to log into the database and execute the desired query. Although you can use any account that has access to the database in question, it is a good idea to create unique user accounts for each application and to provide those accounts only those permissions that are necessary for the application to do its job. Especially if you are working in a multi-database environment, you don't want people to be able to access—let alone change—other users' data.

2.5 ADMINISTRATION TASKS

For relational database systems to work, they need to be installed and set up; databases, tables, and indexes need to be created and maintained; security access needs to be established with usernames and passwords created; other systems support tasks need to be completed. In the following section, we will look at some of the database administration tasks you will need to take care of and show you how to do so.

2.6 CREATING INDEXES

An index, as you would expect, is a data structure that the database uses to find records within a table more quickly. Indexes are built on one or more columns of a table; each index maintains a list of values within that field that are sorted in ascending or descending order.

Getting the data into the database is just the first step in making it available to users. Although the data is searchable once it is inside the database, such searching can be inefficient. Left to its own devices, an RDBMS will go through your tables record by record, matching the query terms against the contents in each of those records. Thus, the more records you have, the more time it will take to look through all of them. You find individual topics within the book and library catalogues provide an index to a library's collection, a database uses its index to match the query parameters and go directly to the appropriate record or records (See figure 2.1).



Figure 2.1

2.7 BACKUPS

A backup server is a type of server that enables the backup of data, files, applications and/or databases on a specialized in-house or remote server. It combines hardware and software technologies that provide backup storage and retrieval services to connected computers, servers, or related devices.

One of the most critical tasks in any computer-based system is to back up the data. Given that computers often get cranky and do things we don't want (or expect) them to do, we need to be able to recover from glitches that occur. To make sure you can do this, you need to have a good backup strategy in place before you begin working with a database management system. When constructing such a plan, you need to keep two types of backups in mind: server and transaction.

2.8 INTERACTION LOGGING

The basic idea behind interaction logging is that as each data maintenance interaction (adding, editing, deleting) is entered into the database, each component SQL query involved in that interaction is written to a logfile. Then, if disaster strikes, all you need to do is to restore from the nightly backup and then enter the interactions from the log file and you will be up and running again.

2.9 SECURITY

Database security refers to the range of tools, controls, and measures designed to establish and preserve database confidentiality, integrity, and availability. This article will focus primarily on confidentiality since it's the element that's compromised in most data breaches. Database security must address and protect the following:

- The data in the database
- The database management system (DBMS)
- Any associated applications
- The physical database server and/or the virtual database server and the underlying hardware
- The computing and/or network infrastructure used to access the database

Database security is a complex and challenging endeavour that involves all aspects of information security technologies and practices. It's also naturally at odds with database usability. The more accessible and usable the database, the more vulnerable it is to security threats; the more invulnerable the database is to threats, the more difficult it is to access and use.

The first thing we need to do is to make sure access to phpMyAdmin is limited to only those persons designated as database administrators. Because phpMyAdmin essentially gives you the "keys to the kingdom," anyone who can access it can do anything they wish to any database in

the system (including deleting a database!). Mechanisms within phpMyAdmin do allow you to provide different persons' administrative access to different databases, but you need to set up some type of access control mechanism to make sure that unwanted users don't get in.

2.10 SELF-ASSESSMENT QUESTIONS

- Q.1 How to select appropriate fields for a library database? Discuss.
- Q.2 Explain data structure and its development phases with examples.
- Q.3 What do you mean by 'establishing user account'? discuss with examples.
- Q.4 Explain the process of administering the database with relevant examples.
- Q.5 Define database backup strategies and set up user security and access controls with examples.

2.11 ACTIVITIES

- i. Design a worksheet of different fields for a university library database.
- ii. Design a worksheet for making users' accounts that can access that data via a Web-based output page.

2.12 REFERENCES

- Baruah, A. (2002). *Library database management*. Gyan Publishing House.
[https://pdf-drive.com/download.php?title=Creating%20Database-Backed%20Library%20Web%20Pages:%20Using%20Open%20Source%20Tools%20\[1ed.\]0838909108,%209780838909102,%209780838998489&content=&downlurl=](https://pdf-drive.com/download.php?title=Creating%20Database-Backed%20Library%20Web%20Pages:%20Using%20Open%20Source%20Tools%20[1ed.]0838909108,%209780838909102,%209780838998489&content=&downlurl=108,%209780838909102,%209780838998489&content=&downlurl=)
[108,%209780838909102,%209780838998489&content=&downlurl=](https://books.google.com.pk/?hl=en)
Krier, L., & Strasser, C. A. (2014). *Data management for libraries: LITA guide*. Chicago: American Library Association. Available at: <https://books.google.com.pk/?hl=en>.
- Preston, C., & Lin, B. (2002). Database technology in digital libraries. *Information Services & Use*, 22(1), 9-17.
- Singh, P. (2004). Library databases: Development and management. *Annals of Library and Information Studies*, 51(I), 72-81.
- Suseela, V. J., & Uma, V. (2017). *Data management for libraries: Understanding DBMS, RDBMS, IR technologies & tools*. Ess Ess Publications.

UNIT-3

INTRODUCTORY PROGRAMMING

Compiled by: **Dr Amjid Khan**
Reviewed by: **1. Dr Pervaiz Ahmad**
2. Muhammad Jawwad
3. Dr Muhammad Arif

CONTENTS

	<i>Page #</i>
Introduction	35
Objectives	35
3.1 What Is Programming?	36
3.2 Basic Concepts	36
Values	36
Variables	36
Arrays	37
Coding	38
Operators	41
Functions	42
Putting It Together Making A Sauce	42
3.3 Function Libraries And Applications	44
3.4 Program Structure	44
3.5 Deciding Which Tool To Use	44
3.6 Self -Assessment Questions	45
3.7 Activity	45
3.8 References	46

INTRODUCTION

Programming is central to any database. This unit covers all aspects related to database programming and applications, programming languages, and open-source language tools. At the end of the unit, self-assessment questions followed by practical activities are given to the students.

OBJECTIVES

After reading this unit, you will be able to:

- Database Programming
- Database reporting
- Database applications
- Programming languages and applications
- Open-source language tools

3.1 WHAT IS PROGRAMMING?

Programming is the implementation of logic to facilitate specified computing operations and functionality. It occurs in one or more languages, which differ by application, domain, and programming model. Programming is simply writing out step-by-step instructions to tell the dumbest entity on the planet (a computer) how to do something. If you start with the assumption that you are smarter than a computer (which, if you're reading this book—or any book—or if you're breathing for that matter, you are), you have already won half the battle! This will involve deciding what you need the computer to do, learning the language (words and grammar) that the computer can understand (because it certainly can't understand English, at least not yet), and patiently (and *that* is the key concept) writing out a step-by-step recipe telling it what to do at each step of the way. We will now look at some of the techniques and concepts we use when telling computers what to do.

3.2 BASIC CONCEPTS

Values

We first need to define several basic concepts we will be using from here forward. We will begin by looking at the ways that values are stored and transmitted within a program, focusing primarily on variables and arrays.

Variables

Variables are the building blocks that contain the individual values the program will use to do its job. They are, if you will, the containers into which you put your raw ingredients with the expectation that they will be transformed into something edible. For example, when creating certain types of sauces, there are three categories of things (variables) you need to have to make the sauce: a fat, a thickening agent, and a liquid. What a cook (programmer) does is to take those variables and assign values to them, depending on what the desired result is. Thus, for a white sauce, the three variables might be given the following values:

```
$fat = "4T butter";  
$thickener = "4T flour";  
$liquid = "8C milk";
```

On the other hand, if a brown sauce were needed, the variables might be these:

```
$fat = "2T oil";  
$thickener = "2T flour";  
$liquid = "3C beef broth";
```

Note that we have created the variables (\$fat, \$thickener, \$liquid) by placing a dollar sign in front of the variable name. This illustrates two principles. The first is the use of the \$ in front. This tells the computer (and the programmer too, for that matter) that the thing \$fat is a variable, not the word fat. Although not all programming languages use a dollar sign to denote a variable, PHP does (as does Perl).

The second principle is that the word I used for a variable describes what the variable is to represent. Although you could call it `$x $big foot`, or even `$mother` of all variables, such a name won't help you as you use the variable in writing the program. Nor are you likely to remember what it represents when you return to the program down the road. By using a variable name that clearly describes the information that it contains (such as `$fat`), you are writing what is known as self-documenting code. This is a concept we use throughout this book.

You may have noticed that, in assigning values to a variable, I place the variable on the left side of the `=` and the value that will be assigned to that variable (poured into that container, if you will) on the right. This is about the closest you will ever come to a universal truth about all programming languages: content to the right of the `=` is assigned to the variable to the left of it.

Arrays

An array is a set of variables that contain related items of information. Arrays differ from variables in that, though variables track individual entities, arrays organize sets of values that go together in some way and that need to be handled together. When we program database searches, we will use arrays to store the individual records before processing them. To return to our cooking metaphor, when creating a dish, you might keep all of the spices together in small bowls on your counter, using them as needed. For example, an Italian dish might have oregano, basil, thyme, marjoram, and garlic.

If you were to create an array to keep the names of all of the spices needed for a dish, you could assign each element individually:

```
$spices[0] = "2T oregano";  
$spices[1] = "4T basil";  
$spices[2] = "4T thyme";  
$spices[3] = "1T marjoram";  
$spices[4] = "3 cloves garlic";
```

Another option is to assign the elements in a single statement:

```
$spices = array("2T oregano", "4T basil", "4T thyme", "1Tmarjoram", "3 cloves garlic");
```

Either way, in using an array, when you need to add the spices, you can refer to them as `$spices[0]`, `$spices[1]`, `$spices[2]`, and so forth instead of using 2T oregano, 4T basil, 4T thyme, and so on. Note that we are using a number as the index (array address, if you will) of the individual elements (or values) of the array. This technique is useful if you will be stepping through the array one at a time and don't need to look for any particular value.

An alternate approach to indexing arrays—associative arrays—is supported by some languages, including PHP. An associative array is one in which we associate a name with the value being stored in the individual element, rather than a number. For example, if we were creating an array of the values needed to make a sauce, we could create it like this:

```
$sauce["fat"] = "4T butter";  
$sauce["thickener"] = "4T flour";  
$sauce["liquid"] = "8C milk";
```

Thus, referring to `$sauce["fat"]` would get us the value 4T butter, `$sauce["thickener"]` would be 4T flour, and `$sauce["liquid"]` would be 8C milk. This technique makes it much easier to access and output items in an array—such as the results of a database search—by using a name that we know (the field name) rather than having to know where in the array a particular value is to be found. The formulation `$result_array[<field_name>]` gives us access to each field of the results. Thus, in handling output from a database search, we can use `$record["title"]` to obtain the title field, `$record["author"]` to get the author, and so on. In this case, by referencing `$sauce["fat"]`, we would obtain the value of 4T butter.

Coding

As noted, creating the program involves writing out each action that needs to be performed to get the desired results. To start, I will use pseudo-code to demonstrate the concept. Pseudo-code is a technique in which you describe what needs to be done in a programming-like way without getting into the details of a particular programming language.

1. `$pan = "4 quart sauce pan";`
2. `$show_hot = "medium-high";`
3. `$heat_source = "stove";`
4. `$sauce["fat"] = "4T butter";`
5. `$sauce["thickener"] = "4T flour";`
6. `$sauce["liquid"] = "8C milk";`
7. Place `$pan` on `$heat_source`;
8. Turn on `$heat_source` under `$pan` to `$show_hot`;
9. `$roux = $sauce["thickener"] + $sauce["fat"];`
10. Place `$roux` into `$pan`;
11. Heat the `$roux`;
12. Heat `$sauce["liquid"]`;
13. `$white_sauce = $roux + $sauce["liquid"];`

Lines 1–6 set the variables for the program. These include the type of pan (`$pan`), the temperature at which to make the sauce (`$show_hot`), what you're cooking on (`$heat_source`), and the array of items with which to start the sauce (`$sauce`). Lines 7–13 then describe the steps to be taken with the variables. When this “program” is run, the program substitutes the contents of each variable for the variable name. Thus, line 7 becomes “Place 4-quart sauce pan on the stove” and line 8 becomes “Turn on the stove under 4-quart sauce pan to medium-high.”

Decision Blocks

Anyone familiar with cooking is aware that the previous recipe does not provide enough information as written. You could perform each step in succession, but the results would be far from satisfactory. For this recipe to work, we need to tell the cook (the computer) certain things, including

- How long should the heating in line 12 continue?
- Does one add `$sauce["liquid"]` all at once?
- How does one know if the process is working correctly?

For each of these questions, instructions need to be included in the program so that the cook (being a computer, not a very bright cook) can proceed properly. The following code shows how we might flesh out the needed information:

```

1. $pan = "4 quart sauce pan";
2. $show_hot = "medium-high";
3. $heat_source = "stove";
4. $sauce["fat"] = "4T butter";
5. $sauce["thickener"] = "4T flour";
6. $sauce["liquid"] = "8C milk";
7. $half_cups = 16;
8.
9. Place $pan on $heat_source;
10. Turn on $heat_source under $pan to $show_hot;
11. $roux = $sauce["thickener"] + $sauce["fat"];
12. Place $roux into $pan;
13. $roux_status = "raw";
14. while ( $roux_status == "raw" ) {
15. stir $roux;
16. $temp = temp( $roux );
17. if ( $temp < 350 ) {
18. print "Not Yet";
19. } else if ( $temp >= 350 && $temp < 400 ) {
20. $roux_status == "cooked";
21. } else {
22. $roux_status = "burned";
23. throw_away( $roux );
24. exit;
25. }
26. }
27. Heat $sauce["liquid"];
28. $bechamel = $roux;
29. for ( $x=0; $x < $half_cups; $x++ ) { 3
30. $bechamel = $bechamel + 1/2 cup ( $sauce["liquid"] );
31. stir $bechamel for 30 seconds;
32. print "You have added $x cups";
33. }

```

In this example, lines 1–7 set the values for each of the variables and lines 9–33 contain the expanded steps. These steps are implemented as decision blocks. Note that this example has three decision blocks—places where the computer decides what to do, whether to continue what it has been doing or to do something new. These three types of blocks (while, if, and for) are those we will use most often in this book. Before examining them more closely, let us take a look at how a block is structured. All decision blocks have the same structure:

```

1. conditional ( condition ) {

```

2. "Do something!";
3. }

There are four things to note here:

- *conditional*—while, if, or for
- *condition*—the condition the computer should check to see if it should execute this block.

The entire block of code is contained in a block—between the { and } characters—that is executed if the condition is met. If the condition is for or while, then it continues to be executed for as long as the condition in line #1 is true (if the condition is if, then the block is run only once). This means that the block starts at the first line and continues executing until the last line before the } character and, if appropriate, checks the condition again at the top of the block to see if it is still true. If so, it goes through the block again. If not, it goes to the first command after the block closes.

Here the { and } characters are used to denote the beginning and end of the block, the way it is done in C, C++, Java, PHP, Python, and other C-like languages. Note that Pascal uses begin/end.

Before continuing, I would like to point out a formatting convention that, though not required, does make programming code a lot easier to read: within the first decision block (lines 15–26), the lines are indented three spaces and, within the blocks within that block (lines 18, 20, and 22–24), the lines are indented again. Although the programming language does not require this, indenting makes it easier to see where decisions are being made, what is being done at each possible point, and where blocks begin and end. I strongly suggest that you follow this practice. Now let us take a closer look at the conditional statements:

while (lines 14–26). This block says, “As long as the \$roux_status is raw, keep on cooking.” Because we initialized—gave an initial value of—raw to \$roux_status at line 13, the first time we get to line 14, it enters the loop. If you don’t do this—or you gave it any other value—it would never enter the while loop at line 14. It then proceeds through to the end of the block (line 26). When it goes there, it goes back to 14 and checks to see if the \$roux_status is still raw. If it is, the program goes through the block again and continues to do so until the condition is no longer true. For this status to change, a test needs to be run each time through the block to see if the status should be changed. This is done in line 16, where the temperature of the mixture is taken. If you do not do this, the status will never change and so the program will never end (you will have entered what is called an infinite loop).

if (lines 17–25). And if the block says “if a condition is true, then, do something once” (where there are multiple possibilities, you can use one or more else if statements to do subsequent checks of the value). In line 16, you get the temperature of the roux and then proceed to your if statements. In this case, the condition is checking the value by taking the temperature of the roux. If the temperature is less than 350, then the first condition if block is entered and the cook is told “Not yet” and the program skips the other two conditions (lines 19 and 21), jumping down to line 25 (the closing of the if/then/else block at which point it returns to line 14, where the status is checked. Because it has not changed, it enters that block again. Also, note that this

block has two more tests. If the temperature is not less than 350, then check to see if the temperature is between 350 and 400. If so, it enters the second block, where the \$roux_status value is set to cook (at which point, it goes to line 26 and then back up to 14 to check the status again. Because the status is now cooked, the program skips down to line 27 (the first line after the while loop) and proceeds to the next task. However, if it finds that \$temp is more than 400, that means that the process has somehow gotten away from you and has burned. You then change the \$roux_status to burned, throw the \$roux away, and exit the program (presumably to start over).

for (lines 29–33). In a for block, you know the number of times you need to go through the block (in this case, the number assigned to a sentinel variable \$half_cups). In the opening statement, you initialize a counter variable, (\$x=0); you tell it to run as long as the counter variable is less than the sentinel variable (\$x<\$half_cups). You then tell it to increment the counter variable by one each time the block is run (\$x++). Then, each time you go through the block, you add 1/2 cup of liquid and stir for 30 seconds, at which point you start the block over and see if your counter variable is now equal to (not less than) your sentinel value, \$half_cups. Once it is equal, that means that all of the liquid has been added and the sauce is made.

When going through a block, it was noted that a counter variable is used to keep track of how many times the block had been executed. This makes the counter variable a valuable resource when used within the block with a numerically indexed array. This is because, as you go through the block, \$x is increasing in value by 1 each time through the block. You could then use \$x to go through an array one at a time, using the variable as the index to the array to access the array values. For example, if you were making an Italian dish, you could access the array as follows:

```
1. $num = count( $spices );  
2. for ( $x=0; $x<$num; $x++ ) {  
3. add $spices[$x];  
4. }
```

The first time through the loop (when the value of \$x was 0), line 3 would read \$spices[0] (whose value is 2T oregano) and it would be executed as:

add 2T oregano

The second time, the value of \$x would be 1 and, because the value of \$spices[1] is 4T basil, it would be executed as:

add 4T basil and so on. This is a technique we will be using often.

Operators

Different programming languages use different operators to denote equal to, greater than, less than, and so on. For example, in PHP, checking to see if \$x and 1 are equal, you would use if(\$x == 1), whereas, in Visual Basic, you would use if(x = 1). You must learn how the language you are using does comparisons.

Functions

The technique involves taking often used code and placing it in a separately named block, called a function. Once the code is in a function, it can be called by invoking the function each time it is needed. Breaking larger tasks into smaller tasks and then writing a function for each smaller task is what is known as structured programming. Structured programming makes writing and debugging a program much easier, increases the flexibility and reuse of code, and reduces the number of times the same routine code appears within a set of programs.

PUTTING IT TOGETHER MAKING A SAUCE

The following code snippet demonstrates this concept by taking the basic sauce creation logic and placing it into a function called `make_sauce()` that takes the basic steps and abstracts them into a stand-alone entity. You will have different ingredients, depending on the type of sauce you want to make. This is where variables prove so handy. What you do is create a set of variables that will contain the basic information you want the program to use when making the desired roux. To change the type of sauce you are making, you merely change the values in the variables passed to the function:

```
1. function make_sauce( $sauce ) {
2. $fat = $sauce["fat"];
3. $thickener = $sauce["thickener"];
4. $liquid = $sauce["liquid"];
5. $pan = $sauce["pan"];
6. $how_hot = $sauce["how_hot"];
7. $done = $sauce["done"];
8. $half_cups = $sauce["half_cups"];
9. $heat_sauce = $sauce["heat_source"];
10. Place $pan on $heat_source;
11. Turn on $heat_source under $pan to $how_hot;
12. $roux = $thickener + $fat;
13. Place $roux into $pan;
14. $roux_status = "raw";
15. while ( $roux_status == "raw" ) {
16. stir $roux;
17. $temp = temp( $roux );
18. if ( $temp < $done ) {
19. print "Not Yet";
20. } else if ( $temp >= $done && $temp < $done + 20 ) {
21. $roux_status == "cooked";
22. } else {
23. $roux_status = "burned";
24. throw_away( $roux );
25. exit;
```

```

26. }
27. }
28. Heat $liquid;
29. $product = $roux;
30. for ( $x=0; $x < $half_cups; $x++ ) {
31. $product = $roux + ( $liquid * 1/2 cup );
32. stir $product for 30 seconds;
33. print "You have added $x cups";
34. }
35. return( $product );
36. }

```

Note that line 1 of the previous example includes the name of the function `make_sauce()` and that immediately following the name, it has the word `$sauce` inside parentheses. Here, `$sauce` is the array of values that are being passed to the `make_sauce()` function. In the program calls `make_sauce()`, an array named `$sauce` is created and is filled with the various pieces of information that the `make_sauce()` function will need to do its work. Note that the sauce array contains eight values. When the calling program is invoked, it in turn sends the `$sauce` array to fill the associated parameters:

```

1. $sauce["pan"] = "4 quart sauce pan";
2. $sauce["$show_hot"] = "medium-high";
3. $sauce["heat_source"] = "stove";
4. $sauce["fat"] = "4T butter";
5. $sauce["thickener"] = "4T flour";
6. $sauce["liquid"] = "8C milk";
7. $sauce["$half_cups"] = 16;
8. $sauce["done"] = 375;
9. $bechamel = make_sauce( $sauce );

```

You may have noticed that line 35 of the function contains the line `return($product)`. Many functions return a result to the calling program. In the case of making gravy, it would not make sense to go to the trouble of sending all of these values off to the `make_sauce()` function if one didn't expect a sauce as the result. This is what the `return(value)` does. It takes the results of the function and returns them to the calling program.

On the other end, line 9 of the calling program contains the line `bechamel= make_sauce($sauce)`. This essentially says: call the `make_sauce()` function, pass it to the `$sauce` array, and place the result in the `$bechamel` variable. Note, if we were making a beef gravy, we would just store different values to the various parts of the `$sauce` array as follows:

```

1. $sauce["pan"] = "iron pot";
2. $sauce["$show_hot"] = "medium-high";
3. $sauce["heat_source"] = "stove";
4. $sauce["fat"] = "8T beef fat";
5. $sauce["thickener"] = "8T flour";

```



```
6. $sauce["liquid"] = "12C beef broth";
7. $sauce["$half_cups"] = 24;
8. $sauce["done"] = 375;
9. $gravy = make_sauce($sauce);
```

3.3 FUNCTION LIBRARIES AND APPLICATIONS

You can take individual functions and place them into a separate file, from which they can be included in any application file in which you need to use them. These libraries are extremely valuable because you do not have to reinvent the wheel. Gathering your functions into library files is only the beginning. Several other developers have also created their function libraries and have made them available on the Internet. By finding and using these libraries, and modifying them for your needs, you can save even more time and trouble. The bibliography lists just some of the libraries available. Beyond just libraries, several development platforms—particularly Perl, PHP, and Python—are being used in the cooperative development of full-blown applications. These can range from library-specific applications such as My Library to more general programs such as shopping baskets, help desk apps, and the like.

3.4 PROGRAM STRUCTURE

When designing any program, you need to take a step-by-step approach to the entire process. Then, within each step will be additional steps that need to be undertaken if the larger step is to be successful. In larger programs, these smaller steps may in turn have steps within them, and so on. This type of structured approach is essential to good program development and one that will be used throughout this book.

3.5 DECIDING WHICH TOOL TO USE

Before proceeding, we should take a look at the various open-source tools available for use in development. All of these tools are also cross-platform—meaning that you can run them on Windows, Unix/Linux, and Mac OS X computers. Although PHP is the tool that we will be spending the most time on, it is by no means the only one you can use. You can find strong proponents of any of these tools, each person giving strong reasons why their favourite is the “mother of all development tools.” Although each tool (including PHP) has certain strengths and weaknesses, we won’t go into those details here:

There are several points to consider in deciding which tool to use:

- *Ease of learning.* How long does it take to get up to speed with a language? Does it require specialized skills?
- *Maintainability.* How easily can the applications written in the language be supported and maintained by the developers or by others? How expensive will it be to hire/support somebody to work in the language?
- *Readability.* Does the syntax make it easy to understand what is going on in the code? Does the language use idiosyncratic operators that are hard to understand or that are inconsistent in their meaning?

- *Rapid development support.* Does it make it easy to develop applications quickly? Do you need to spend a lot of time writing and testing code?
- *Error handling.* Does the tool have mechanisms that will allow you to gracefully handle unexpected events and problems, providing useful information to developers and, for public applications, appropriate messages to users?
- *Cross-platform.* Does it support more than one type of server? Can it run on multiple operating systems?
- *Built-in functions.* Does it have a wide variety of functions built into the system that you can use or is it up to you to write them yourself (or find someone who has already done so)?
- *Flexibility and extensibility.* Will it handle all the things you need to have it do? Does the language allow for the creation of function libraries or modules?
- *Performance.* How well does it perform, particularly given multiple users?
- *User base.* How popular is it and how many people are using it? In and of itself, this is not important. However, the more users there are, the more likely it is to be developed further; the more people out there who can get you through problems; and the more likely you are to find people who can program in it.
- *Libraries, modules, and third-party applications.* Are there libraries and other code out there that you can obtain and add to your application (thereby avoiding having to reinvent the wheel)?

There are essentially seven open-source language tools you can use to develop your project. The first two—gcc/g++ and Java—are industrial-strength, professional programming languages. The other five—TCL, Perl, PHP, Python, and Ruby—are essentially Web server-based scripting languages that allow Web development without demanding the computational or intellectual overhead of the first two. These can be used via the CGI interface and each—to varying degrees—can be integrated into the Apache server (and some with other Web servers) as an included module. In addition, except for tcl, all five include object-oriented programming (OOP) capabilities, thus making it more likely for you to be able to find OOP libraries that will make your development work easier.

3.6 SELF-ASSESSMENT QUESTIONS

- Q.1 Define database programming and its basic concepts with examples.
- Q.2 Write a note on programming languages and applications.
- Q.3 Report writing is central to any database-backed Web page, how and why?
- Q.4 Discuss with examples open-source language tools.

3.7 ACTIVITY

- Design a project for an academic library with the help of open-source language tools.

3.8 REFERENCES

- Baruah, A. (2002). *Library database management*. Gyan Publishing House.
- Krier, L., & Strasser, C. A. (2014). *Data management for libraries: LITA guide*. Chicago: American Library Association. Available at: <https://books.google.com.pk/?hl=en>
- Preston, C., & Lin, B. (2002). Database technology in digital libraries. *Information Services & Use*, 22(1), 9-17.
- Singh, P. (2004). Library databases: Development and management. *Annals of Library and Information Studies*, 51(I), 72-81. Available at: <http://nopr.niscair.res.in/bitstream/123456789/7488/1/ALIS%2051%282%29%2072-81.pdf>
- Suseela, V. J., & Uma, V. (2017). *Data management for libraries: Understanding DBMS, RDBMS, IR technologies & tools*. Ess Ess Publications.
- Westman, S. R. (2006). *Creating database-backed library web pages: Using open source tools*. Chicago: American Library Association. Available at: [https://pdfdrive.com/download.php?title=Creating%20DatabaseBacked%20Library%20Web%20Pages:%20Using%20Open%20Source%20Tools%20\[1ed.\]0838909108,%209780838909102,%209780838998489&content=&downlurl=](https://pdfdrive.com/download.php?title=Creating%20DatabaseBacked%20Library%20Web%20Pages:%20Using%20Open%20Source%20Tools%20[1ed.]0838909108,%209780838909102,%209780838998489&content=&downlurl=)

UNIT-4

CREATING REPORTS

Compiled by: **Dr Amjid Khan**

Reviewed by

- 1. Dr Pervaiz Ahmad**
- 2. Muhammad Jawwad**
- 3. Dr Muhammad Arif**

CONTENTS

	Page #
Introduction	49
Objectives	49
4.1 Report Writing	50
4.2 Creating A Basic Report (Report Structure)	50
Hard Coding	55
Action Page	58
Inputting Multiple Values Using And	59
Drop-Down Lists And Keyword Searching	61
using and or	63
4.3 Self-Assessment Questions	63
4.4 Activity	63
4.5 References	64

INTRODUCTION

Report writing is important to any database-backed Web page, whether it's a stand-alone page or a complex search screen within a large application. Any time you create a query, send it to the database and then format the returned results, you are creating a report. There are three basic methods you can employ to create a report. In this unit, we show you a variety of reports that you can create using PHP. We will begin by creating a simple page to demonstrate the basics of writing a Web-based report. Next, we examine how to create pages using search parameters hard-coded in the page, by passing values to the page via the URL, and then by using input from a form. We will then build on that knowledge to create searching applications. At the end of the unit, self-assessment questions followed by practical activities are given to the students.

OBJECTIVES

After reading this unit, you will be able to:

- Design database search interface
- Create a basic report (report structure) and create the report program
- Develop hard coding and passing parameters via URLs
- Know how to search the input form and action page?
- How to create basic and advanced searching applications

4.1 REPORT WRITING

Report writing is important to any database-backed Web page, whether it's a stand-alone page or a complex search screen within a large application. Any time you create a query, send it to the database and then format the returned results, you are creating a report. There are three basic methods you can employ to create a report.

- First, you can use proprietary tools. Some database applications, such as FilemakerPro and ProCite (as well as third-party tools such as CrystalReports) provide built-in tools to query a database and output the results without any need for programming on the user's part.
- A second approach is CGI programming. Here, stand-alone programs written in a traditional programming language are used to create the entire report.
- In the Web server-based scripting approach, the Web server has been outfitted with special modules to handle database requests and other types of programming tasks. Programming instructions are embedded directly into an HTML-like page that is given a special extension (such as search.php rather thansearch.html). When the user requests this special page, the Web server notes the special extension and then sends the request to the appropriate module. The module then processes the page, running the code inside the code areas as needed and returning the results to the server where they are in turn passed onto the user. The beauty of this approach is that the languages involved are easier to use and (because you don't have to program the HTML sections) easier to develop and that the module (being part of the Web server) performs its task quickly.

As you can tell from these descriptions, the approach we will be using involves programming. Therefore, it is a good idea to begin exploring how you (yes, YOU) can become a programmer.

4.2 CREATING A BASIC REPORT (REPORT STRUCTURE)

In our first example, we will create a phone list from the database. The structure of this program is fairly simple, and is broken down into four parts:

- *Connecting to the database*—establishing a connection by sending the database in question a username and password (required by most RDBMS products)
- *Creating and sending a query*—writing a properly formed SQL query and then submitting it to the RDBMS
- *Creating the Web page*—taking the results you receive back from the database and using them to create an HTML page
- *Outputting the results*—the Web server sends the resulting page back to the user

Figure 4.1 graphically represents this process (moving top to bottom and left to right). We have broken down the sample examples into three steps—connecting to the database, creating and running the query, and creating the Web page (as noted, the Web server takes care of the fourth step)—to make what we are doing more clear.

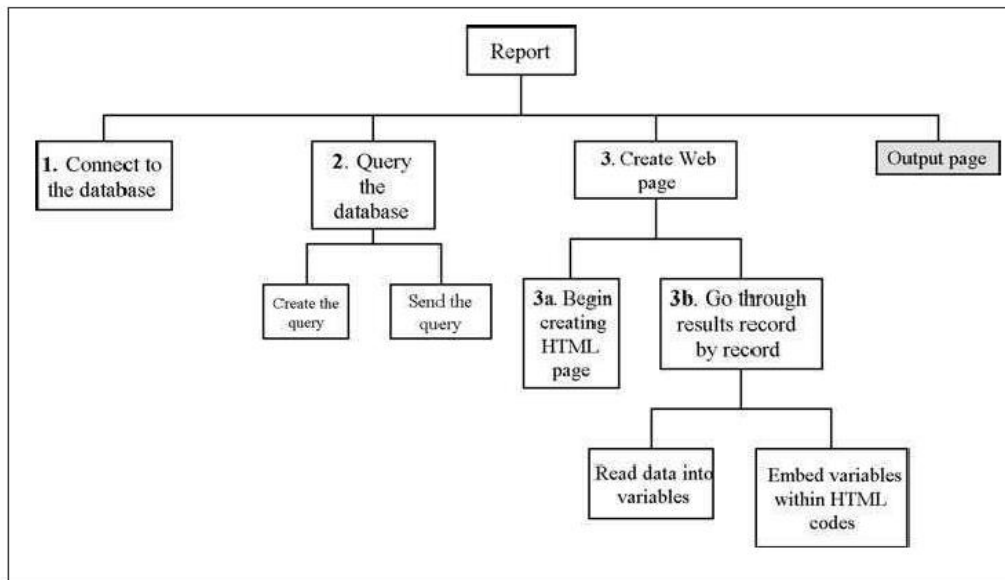


Figure 4.1. Basic report structure

4.3 CREATING THE REPORT PROGRAM

Let's begin by using this structure to create a report to output data from the telephone database. In this study guide, we use four types:

- `#` Placing this character in the first position on a line causes the line to be skipped. We use this character, along with full lines of `#` before and after the comments to help them stand out. For example, the major sections of code we discussed above will have such comments at the beginning of each section.
- `/* .. */` Placing `/*` on the line before a comment, `*/` on the line after, and then placing an asterisk in the first position on each line in between identifies the block to the PHP module as a comment.

(The asterisks at the beginning of intermediate lines are not required but do help to distinguish them from lines of code to the reader.) We use this style of comment to explain subunits within the major sections of the pages.

- `//` Double forward slashes anywhere in a line will cause the PHP module to ignore everything after it on that line. For that reason, we will be using it to explain individual lines of PHP code, also known as inline comments. For example, when the PHP engine comes to `$x = $x+1; // add one to $x` it will compute `$x = $x+1` and then skip to the next line.
- `<!-- -->` We use traditional HTML commenting style for comments in HTML areas. When we run the script, the code creates a page that looks like the one in figure 4.2.



Figure 4.2. Creating the report program

Let's examine this script to see how it works. First, we set up a PHP area by placing the opening PHP tag (`<?PHP`) on line 1 and the closing tag (`?>`) on line 21. Anything that we type in between these two tags will be interpreted as PHP code. Within this area, we will undertake the tasks listed in the first two boxes listed in figure 4.1: we make the connection to the database and send the query to the server. First, we make the connection in lines 13–14, using the parameters we set in unit 2, as shown in example 4.1.

Example 4.1

```

13 $db = mysql_connect( "localhost", "Phones", "Fone_Usr" );
14 mysql_select_db( "web_info", $db );

```

Line 13 uses PHP's `mysql_connect()` function to connect to the database server using three parameters: the host (in this case, `localhost`), the username to connect as (the one we created in chapter 3, `Phones`), and the username's password (`Fone_Usr`, also set up in unit 2), assigning the resulting connection to `$db` (note that if you changed either the username or password in creating the database, you will need to replace what is here with what you used at that time).

Then, in line 14, we use PHP's `mysql_select_db()` function to use that connection to tell the MySQL database server that we want to use the `web_infodatabase` and assign that connection to the `$db` handle (a handle is essentially available that you use to communicate with the database). Next, we create our SQL query and send it to the MySQL database server. This is done in lines 19 and 20 (see examples 4-2).

Example 4-2

```
19 $query = "SELECT * FROM phones ORDER BY last_name,first_name";
20 $result = mysql_query( $query, $db ) or die( mysql_error() );
```

Here we create a variable name `$query` and assign it an SQL query that asks for all fields (`SELECT*`) from the `phones` table (`FROM phones`), sorted by `last_name` and, within that, the `first_name` (`ORDER BY last_name,first_name`). Then in line 20, we use PHP's `mysql_query()` function to send the `$query` to the database (via `$db`), storing the results to `$result`. Once we have our data, we proceed to the next step and create the actual report page. First, we define the HTML area into which we will embed our search results in lines 28–34 (see example 4.3).

Example 4.3

```
28 <html>
29 <head>
30 <title>Phone Directory</title>
31 </head>
32 <body>
33 <center><h1>Phone Directory</h1></center>
34 <table border="1" width="100%">
```

Next, we read the results into variables and output the values embedded within HTML codes. We do the first in lines 43–47 (see example 4-4).

Example 4.4

```
41 <?php
42 while ( $row = mysql_fetch_array( $result ) ) {
43     $name = $row["last_name"] . ", " . $row["first_name"];
44     $phone = $row["phone"];
45     $department = $row["department"];
46     $location = $row["location"];
47     $email = $row["email"];
48 }>
49 <tr>
50     <td><?php echo "$name" ?></td>
51     <td><?php echo "$phone" ?></td>
52     <td><?php echo "$department" ?></td>
53     <td><?php echo "$location" ?></td>
54     <td><a href="mailto:<?php echo "$email" ?>@mylib.edu"><?php echo "$email" ?>@mylib.edu</a></td>
55 </tr>
56 <?php
57 }
58 ?>
```

One of the techniques to output arrays is a while block, permitting us to access the contents one item at a time. What this block is saying is this: While there is still a record to process, use PHP's `mysql_fetch_array()` to extract one record—in the form of an array—and store it to an associative array named `$row`. Then, just as we did with the `$sauce` array in unit 3, we can take each element in the `$row` array and save it to a variable for output. Using `mysql_fetch_array()` to give us associative arrays (as described in chapter 4) allows us to use the field name to access the field's value. Next, we go through each field we want to output and save its value to a variable with the field's name. In line 43, we place two fields into the first variable (`$name`): the `last_name` and `first_name` fields so that we can treat the name as a single entity. We do this by joining them (concatenating) by taking the `$name` variable and

- assigning the value of the `last_name` element of `$row` to it
- concatenating (adding) a comma and space at the end of `$name`
- tacking on the `first_name` from `$row` to the end of `$name`

The final task is to output the values on our Web page. Although there are several ways to do this, here we temporarily break out of the PHP block (line 48) and output the variables using HTML tags (lines 49–55). Then, so that PHP doesn't become confused as to where things end, we close the while block we began in line 41 in lines 56–58 (opening up a PHP block so that the brace will be interpreted as a PHP curly brace).

Note that lines 49–55 are not within a PHP area. We therefore use `echo` with each variable within a PHP block (`<?PHP echo $variable ?>`) to print out the actual value. While we could have remained in the PHP block, that would have required more typing (using `echo` for the HTML tags). This technique of using `<?PHP echo $variable ?>` can be very useful and one that we will be using quite a bit in accessing PHP variables inside HTML areas. One additional thing I have done to make this list more useful is to turn the e-mail address into

an actual mail to link in line 54 in example 4.4. We do this by simply wrapping the output for the \$email variable in the appropriate HTML code.

Obtaining Selective Output

One of the advantages of using a database is that it allows you to retrieve only those records that match what you want to retrieve. As you may recall from unit 1, we create those subsets by adding a WHERE condition to the query in the form WHERE <fieldname> = '<condition>'. Then, when the database is searched, the search engine filter retrieves only those records that match that condition.

Hard Coding

There are three ways you can set the condition. One way is to hard code the query. For example, say you want to create a staff page for the Music Library. You would simply go into the PHP script above and change lines 20–22 to read as shown in example 4.5. You would then get the screen shown in figure 4.3.

Example 4.5

```
20 $query = "SELECT * FROM phones
21         WHERE department='Music'
22         ORDER BY last_name,first_name";
23 $result = mysql_query( $query, $db ) or die( mysql_error() );
```

Phone Directory - Mozilla

File Edit View Go Bookmarks Tools Window Help

Back Forward Reload Stop

http://localhost/examples/Chapter_5/mus_lib.php Search Print

Home Bookmarks Instant Message WebMail Radio People Yellow Pages Download Calendar

Phone Directory

Baker, Julius	3-3278	Music	150 Mahler Center	jbaker@mylib.edu
Bell, Joshua	7-9898	Music	150 Mahler Center	jbell@mylib.edu
Brain, Dennis	3-6849	Music	150 Mahler Center	dbrain@mylib.edu
Brannigan, Owen	6-4440	Music	150 Mahler Center	o-brannigan@mylib.edu
Elliott, Willard	3-1933	Music	150 Mahler Center	welliott@mylib.edu
Fox, Virgil	3-4984	Music	150 Mahler Center	vfox@mylib.edu
Horenstein, Jascha	3-2292	Music	150 Mahler Center	jhorenstein@mylib.edu
Mack, John	3-2333	Music	150 Mahler Center	jmack@mylib.edu
Marcellus, Robert	3-9233	Music	150 Mahler Center	rmarcellus@mylib.edu
Russell, Anna	7-7463	Music	150 Mahler Center	arussell@mylib.edu
Sapp, Alan	3-1766	Music	150 Mahler Center	asapp@mylib.edu
Taylor, Deems	3-9222	Music	150 Mahler Center	dtaylor@mylib.edu

Figure 4.3. Coding for phone directory

Passing Parameters via URLs

Besides hard coding a condition into a script, we can also pass a parameter to the page via the URL and the page can then use it as the basis for its query. An extremely useful technique, this enables us to automatically generate a page containing a defined subset of a database by simply passing the page a parameter that defines that subset. By allowing us to encode this search into a URL, we have much more flexibility in how we use our database. For example, say we were responsible for the Music Library's Web pages and wanted to include a link to a Music Library staff directory on our department home page. We could create such a link by embedding the following URL on that page:

`Directory`

To process this request, we need to copy the report.php to url_report.php and then make appropriate changes to url_report.php: Because variables passed via URLs are passed using the GET method, we first make sure that such a value has been passed (see line 20 of example 4.6). If so, we save it to a variable named \$department (line 21). If not, we indicate to the user to provide a value and how to do so (lines 24–25) and exit (line 26).

Example 4.6

```
20 if ( isset( $_GET["department"] ) ) {
21     $department = $_GET["department"];
22     $department = urldecode($department);
23 } else {
24     echo "You will need to provide the name of a department in the form ";
25     echo "<b><i>url_report.php?department=<i>department name</i></i></b> as the URL";
26     exit;
27 }
28 $query = "SELECT * FROM phones
29          WHERE department='{$department}'
30          ORDER BY last_name,first_name";
31 $result = mysql_query( $query, $db ) or die( mysql_error() );
```

Note that some of the departments in the department list have two words in their name. However, we can't create a URL string containing a space in the middle of it (for example, `<ahref="url_report.PHP?Department=SpecialCollections">SpecialCollections`) because the browser won't process anything after the first space it encounters. Therefore, a URL for this resource must replace spaces with %20 (the hexadecimal code for space): `SpecialCollections`. However, we need to turn it back into space before sending it to the database. Therefore, in line 22, we use PHP's URL decode () function to do that transformation for us. Finally, we need to modify the query so that it uses the parameter that has been passed to it (line 29).

Before proceeding, I would like to briefly explain the concept of superglobals. Superglobals(reserved variables) are global associative arrays built into PHP to make

certain types of information available between PHP pages. In the case above, `$_GET` is an array that contains all of the information being passed to the action page via the GET method. There are a number of these superglobals we will be dealing with throughout this book. Although we won't spend a lot of time exploring variations of embedding search parameters within a URL, it is an extremely powerful and useful tool in web database development. It allows you to pass parameters via the URL, permitting you to create canned searches of your database and embedding them as hyperlinks either on Web pages or, if appropriate, in the 856 fields of MARC bibliographic records in your online catalogue.

Search Input Form

To enable end-user searching of a database, we need to have two files: one a form into which users can enter their search terms (an input form) and one that will take those search terms, query the database, and output the results (action page). The first thing we do is create the query form (`report_query.php`). Although it is fairly straightforward, there are a couple of aspects we should take note of. First, in line 7 (example 4.7) we have defined the action (the name of our action page) as `report_search.php`. The other is that, in line 23 (example 4.8), we have created a text `<input>` box with the name of a department.

Example 4.7

23

```
<input type="text" name="department" size="30" maxlength="80">
```

Example 4.8

7

```
<form method="POST" action="report_search.php">
```

This name here will be the name of the variable that will be passed to the action page. We use the name of the field (with the same capitalization) that will be searched. As you will see, doing this reality facilitates your work with databases and makes your code much more supportable. Although this page has a `.php` extension, it does not contain any actual PHP code. Creating HTML pages with the PHP extension opens the door to some possibilities, including:

- pages with global variables, headers and footers, and other features that PHP can make available to you

PHP-based authentication and authorization mechanisms to restrict access to the pages (something not possible with `.html` extension files).

Action Page

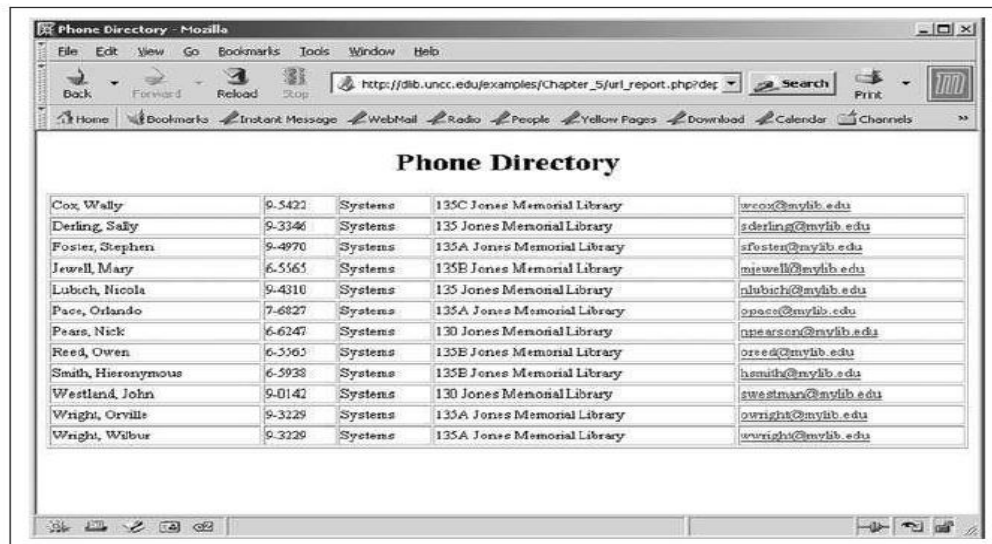
The following figure demonstrates the basic structure of an action page in a searching application. As you can see in figure 4.9, it is a slight variation on the report structure discussed above, the only difference in getting the user input.

To create our action, we take `url_report.php` and copy it to `report_search.php`. Then, because we have defined the method in `report_query.php` as POST, we need to modify line 20 (example 4.9) to use the `$_POST` array instead of the `$_GET` array we were required to use when taking values from a URL string.

Example 4.9

```
20 $department = $_POST["department"];
```

Now, if we enter **systems** into the **Department** text box in the `$report_query` search inputting form, we get the result shown in figure 4.5. Before proceeding, let me point out a naming convention used here that will be used throughout the book. All input forms meant to gather user terms (and then pass them on to a searching action page) will have a base name of `query.php` with an identifier (in this case, `report_`) as a prefix to indicate its role in the application. The action pages that do the search and output results will use `search.php` (with the same identifier prefix) as their base name. This practice allows you, first, to group pages based on what part they play in the greater application and, second, to have a predictable way of naming files so that you will be able to tell by looking at the name what each one of them does.



Name	Phone Number	Department	Address	Email
Cox, Wally	9-5422	Systems	135C Jones Memorial Library	wcox@mylib.edu
Dering, Sally	9-3346	Systems	135 Jones Memorial Library	sdering@mylib.edu
Foster, Stephen	9-4970	Systems	135A Jones Memorial Library	sfoster@mylib.edu
Jewell, Mary	6-5565	Systems	135B Jones Memorial Library	mjewell@mylib.edu
Lubich, Nicola	9-4310	Systems	135 Jones Memorial Library	nlubich@mylib.edu
Pace, Orlando	7-6827	Systems	135A Jones Memorial Library	opace@mylib.edu
Pears, Nick	6-6247	Systems	130 Jones Memorial Library	npearson@mylib.edu
Reed, Owen	6-5565	Systems	135B Jones Memorial Library	oreed@mylib.edu
Smith, Hieronymus	6-5938	Systems	135B Jones Memorial Library	hsmith@mylib.edu
Westland, John	9-0142	Systems	130 Jones Memorial Library	jwestland@mylib.edu
Wright, Orville	9-3229	Systems	135A Jones Memorial Library	owright@mylib.edu
Wright, Wilbur	9-3229	Systems	135A Jones Memorial Library	wwright@mylib.edu

Figure 4.5

Creating Searching Applications

Searching multiple tables is more complex because it involves joins to search. It is better to start slowly and cover the basic elements of searching applications. One possible alternative is to write report programs that extract information from the database and store it in individual files. You would then use a Web search tool, such as Swish-e, to index and search those files (rather than the database directly).

Inputting Multiple Values Using AND

To search multiple terms, we need to provide a mechanism by which multiple terms can be entered. Then we need to take the search parameters and construct appropriate SQL queries. In the following section, some examples are given for doing just that.

Multiple Text Boxes

In the following example, we will construct a search form (multi_query.php) that will allow the user to search four fields: first name, last name, department, and location. Although the form is straightforward, when writing the action page (multi_search.php), we need to take into account that the user may fill in any, all, or even none of the fields on the page. We then need to take appropriate action based on their input. Example 4.9 demonstrates the last of the three cases: the user has input nothing. Because we need to have something to search for, at least one of the fields needs to have a value.

Example 4.10

```
19 $first_name = trim( $_POST["first_name"] );
20 $last_name = trim( $_POST["last_name"] );
21 $department = trim( $_POST["department"] );
22 $location = trim( $_POST["location"] );
23
24 /*****
25  * Next, we construct the $where string to check each one of the form's
26  * fields. If it finds a value, it adds an appropriate WHERE condition
27  * to the $where_str
28  *****/
29 if ( ($first_name == "" && $last_name == "" && $department == "" && $location == "" ) ) {
30     echo "Please enter a query";
31     exit;
32 }
```

What the block between lines 29 and 32 is essentially saying is that if first_name is blank and last_name is blank and the department is blank and location is blank (&& being MySQL's Boolean AND operator), then tell the user "Please enter a query" and exit the program. We can assume that the program will skip this block if any of the fields are not blank, and go on to line 55 (see example 4.11).

To make this work, we need to make sure that there are no extraneous spaces in the queries, such as the user accidentally hitting the space bar at the end of their input. We do this by using PHP's trim() function to eliminate any spaces before or after the terms in lines 19–22 when we read in the query parameters.

Example 4.11

```

34 $w = 0;
35
36 /*****
37  * Then, we see if the second search has a field and value.  If so, we attach
38  * it to the end of the $where string.
39  *****/
40 if ( $first_name != "" ) {
41     $where_ary[$w] = "first_name = '$first_name'";
42     $w++;
43 }
44
45 /*****
46  * Then, we do the same with the Last_Name
47  *****/
48 if ( $last_name != "" ) {
49     $where_ary[$w] = "last_name = '$last_name'";
50     $w++;
51 }
52
53 /*****
54  * Then, we do the same with the Department.
55  *****/
56 if ( $department != "" ) {
57     $where_ary[$w] = "department = '$department'";
58     $w++;
59 }
60
61 /*****
62  * Then, we do the same with the Location.
63  *****/
64 if ( $location != "" ) {
65     $where_ary[$w] = "location = '$location'";
66     $w++;
67 }

```

The next section of code (lines 34–66), shown in example 4.11, checks each variable coming from the form to see if the user has input something for that field. For each one it finds, it adds a WHERE condition to an array (\$where_ary) of WHERE conditions. In the first one, it checks to see if \$first_name is blank. If not, it saves first_name = '\$first_name' to the \$where_ary[\$w] element (\$w being equal to 0 at that point, having been set to that value in line 34) and then adds 1 to \$w (making it equal to 1). The program then goes through the rest of the list of fields, adding an element to the \$where_ary array and adding 1 to \$w for each field for which a value has been entered by the user. When we're through, we have an array of where conditions to add to the end of our SQL query.

At line 71 (see example 4.12), we begin for a block that will go through the \$where_ary array one element at a time and add its content to the \$where statement. In line 72, the program checks to see if this is the first element to be processed. The reason for this is that we need to have an AND between all elements (but not before the first one). By processing the first condition or element separately, we can place the condition field = 'value' at the beginning of the WHERE statement. Then, for every statement that comes afterwards, we just tack AND field = 'value' onto the end of the statement (as we do in lines 74–76)—thereby allowing us to not add more ANDs than we need.

Example 4.12

```
68  /*****
69  * Now we go through the $where_ary, constructing our $where statement.
70  *****/
71  for ($x = 0; $x < $w; $x++) {
72      if ( $x == 0 ) {
73          $where = $where_ary[0];
74      } else {
75          $where .= " AND $where_ary[$x] ";
76      }
77  }
78
79  /*****
80  * Now we put all of these things together, create our query and send it
81  * to the database
82  *****/
83  $query = "SELECT * FROM phones
84          WHERE $where
85          ORDER BY last_name,first_name";
86  $result = mysql_query( $query, $db ) or die( mysql_error() );
87  $num_rows = mysql_num_rows( $result );
```

Once we have built a \$where statement, we use it in constructing our query(lines 83–85) and send it to the database (line 86). Then, in line 87, we use PHP'smysql_num_rows() function to store the number of records retrieved by our query to \$num_rows. If the number is 0, we inform the user in lines 109–112(example 4-13) that their search has been unsuccessful and that they should try again (and exit the program). If it is not, then we proceed to create the screen as in earlier examples.

Once we have built a \$where statement, we use it in constructing our query(lines 83–85) and send it to the database (line 86). Then, in line 87, we use PHP'smysql_num_rows() function to store the number of records retrieved by our query to \$num_rows. If the number is 0, we inform the user in lines 109–112(example 4.13) that their search has been unsuccessful and that they should try again (and exit the program). If it is not, then we proceed to create the screen as in earlier examples.

Example 4.13

```
109  if ( $num_rows == 0 ) {
110      echo "Your search retrieved no results. Please go back and try again";
111      exit;
112  }
```

Drop-Down Lists and Keyword Searching

DROP-DOWN LISTS: There may be times when your users don't know the alternatives or the values in the database. In such cases, it can be helpful to give them a list from which they can select, rather than making them guess what may be in there. HTML forms provide a wonderful technique for doing that: the select list. Example 4-14 modifies the previous

form to make a list of what is in the database and to create an alphabetized drop-down list from which the user can select.

Example 4-14

```
22 Department: <br>
23 <select name="department">
24   <option></option>
25 <?php
26   $db = mysql_connect( "localhost", "Phones", "Fone_Usr" );
27   mysql_select_db( "web_info", $db );
28   $query = "SELECT DISTINCT department FROM phones ORDER BY department";
29   $result = mysql_query( $query, $db ) or die( mysql_error() );
30   while ( $row = mysql_fetch_array( $result ) ) {
31     $department = $row["department"];
32     echo "<option name=\"{$department}\">{$department}</option>\n";
33   }
34   ?>
35 </select>
```

In lines 22–35 of `multikeyword_query.php`, we can see how this can be achieved. In this example, we treat the department field in this way by creating a select list and filling it with data from the database. Lines 22–24 and 35 provide the HTML structure for the list. In lines 25–34, we create a PHP block where we do a mini-report containing all three sections of a full report: making a connection to the database (26–27), creating the query (28), running the query (29), and then embedding the results in HTML, this time as a set of `<option>` tags(30–33).

KEYWORD SEARCHING: Another feature we will want in a searching application is to be able to search by keyword rather than requiring the user to search by the entire contents of the field. There are three ways to implement keyword searching in MySQL:

1. Using the `LIKE` operator, is an unsatisfactory technique to use because it often returns counterintuitive results. It also takes truncation out of the hands of the end-user.
2. Use MySQL's `FULLTEXT` indexing capabilities. Although a proprietary technique within MySQL, many other products do offer similar capabilities.
3. Use regular expressions. Regular expressions are a very powerful—if not easily learned—technique built into many systems, including MySQL.

Using and OR

All the searches we have done until now have placed a Boolean AND between the terms we have been searching. Although using AND can help refine your search, there may be times you want to expand your results set by OR'ing terms together. Before proceeding, you need to go through what is known in programming (and other) circles as operator precedence.

4.3 SELF-ASSESSMENT QUESTIONS

- Q.1 Explain the basics of a database search interface with examples.
- Q.2 Describe the steps involved in creating the reporting program. Also, explain the structure of creating a basic report with suitable examples.
- Q.3 How to create basic and advanced searching applications? Explain with the help of examples.

4.4 ACTIVITY

- Study this unit carefully and design a database search interface for a library.

4.5 REFERENCES

- Baruah, A. (2002). *Library database management*. Gyan Publishing House.
- Krier, L., & Strasser, C. A. (2014). *Data management for libraries: LITA guide*. Chicago: American Library Association. Available at: <https://books.google.com.pk/?hl=en>
- Preston, C., & Lin, B. (2002). Database technology in digital libraries. *Information Services & Use*, 22(1), 9-17.
- Singh, P. (2004). Library databases: Development and management. *Annals of Library and Information Studies*, 51(I), 72-81. Available at:
<http://nopr.niscair.res.in/bitstream/123456789/7488/1/ALIS%2051%282%29%2072-81.pdf>
- Suseela, V. J., & Uma, V. (2017). *Data management for libraries: Understanding DBMS, RDBMS, IR technologies & tools*. Ess Ess Publications.
- Westman, S. R. (2006). *Creating database-backed library web pages: Using open source tools*. Chicago: American Library Association. Available at:
[https://pdfdrive.com/download.php?title=Creating%20DatabaseBacked%20Library%20Web%20Pages:%20Using%20Open%20Source%20Tools%20\[1ed.\]0838909108,%209780838909102,%209780838998489&content=&downlurl=](https://pdfdrive.com/download.php?title=Creating%20DatabaseBacked%20Library%20Web%20Pages:%20Using%20Open%20Source%20Tools%20[1ed.]0838909108,%209780838909102,%209780838998489&content=&downlurl=)

UNIT-5

PROJECT DESIGN

Compiled by: **Dr Amjid Khan**

Reviewed by **1. Dr Pervaiz Ahmad**
2. Muhammad Jawwad
3. Dr Muhammad Arif

CONTENTS

	Page #
Introduction	67
Objectives	67
5.1 An Overview	68
5.2 Defining-Project	68
5.3 Describing The Data Model	69
Collecting all the Pieces.....	69
5.4 Building The Model	70
5.5 Designing The Application	73
5.6 Application Flow.....	73
5.7 Adding A Record	73
5.8 Editing A Record	74
5.9 Deleting A Record.....	74
5.10 Views—Bringing Data and Application Together.....	75
5.11 Defining The Views	75
5.12 Defining Tasks	79
5.13 Interface Design—Creating The Screens.....	81
5.14 Putting It Together—Public Interfaces	82
Pages by Subject	82
Public Searching Interface	82
Testing Procedures.....	82
5.15 Formalizing The Process.....	82
5.16 Self-Assessment Questions	83
5.17 Activity	83
5.18 References	84

INTRODUCTION

The goal of the design process should be a set of specifications that detail the formal requirements and technical specifications of the proposed application, provide a roadmap that programmers can use to code the application, and serve as the basis for documenting the system once it is in place. This unit describes the database project and its design, different views, and tasks of the project model. This unit also explains designing a public searching interface and testing procedures. At the end of the unit, self-assessment questions followed by practical activities are given to the students.

OBJECTIVES

After reading this unit, you will be able to:

- Explain the database project and its design.
- Build the data model and design the application.
- Describe the views and tasks of the project.
- Design public searching interface and testing procedures.

5.1 AN OVERVIEW

The goal of the design process should be a set of specifications that detail the formal requirements and technical specifications of the proposed application, provide a roadmap that programmers can use to code the application, and serve as the basis for documenting the system once it is in place. These specifications should be the result of discussions with the user or users for whom the proposed application is being built. It is critical to undertake this planning process before attempting to implement a project. If you don't, you may run into significant problems down the road.

The same sort of process should be used when building a database application. You, as the architect (developer), need to work with the future users to define what they want in the application, usually in stages:

- Define the goal and purpose of the project.
- Determine, from users and others, what data to include in the application.
- Define how the application will work and how the data fit into the application.
- Present these to the users for feedback, then incorporate their input with your research to develop a model.
- Based on this feedback, take what they have given you, further refine your ideas, and make appropriate changes.
- Repeat steps 4–5 until you reach an agreement.
- Finalize design documents and have all parties sign off on them.

Following these guidelines and procedures should greatly facilitate the process and help avoid unneeded and costly delays due to having to add features after the design phase is finished.

5.2 DEFINING THE PROJECT

When starting, the first thing to do is to gather information about the proposed project. This involves the developer getting together with the user or users for whom the database application is being developed and asking questions about the current environment, needs, and the types of tasks for which the proposed program will be used. Questions can include: Is there a preexisting database (either paper or computer-based) that could be used as a model? to obtain data? What sorts of outputs are desired? Web pages? reports? EAD documents? Are there administrative functions that it will need to perform? If so, what sorts of safeguards are needed for the data? Is a database even the appropriate tool for the job? It is important to ask such questions at the beginning. Although a project may sound simple when proposed, the devil, as they say, is always in the details. Not only is having this information useful when developing the application, but it also helps in planning and prioritizing the project. Not all the questions need to be asked during the first meeting. In some cases, when one has multiple potential projects, you may wish to fill in

only those sections that allow developers and administrators to compare the relative importance, costs, and timeframes of different projects, thereby enabling them to prioritize.

5.3 DESCRIBING THE DATA MODEL

Once you have the go-ahead to start the project, you need to begin designing the application. The first thing that the developer should do is to take the ideas collected in the initial interviews and begin building a data model. Although a fancy-sounding concept, a data model is essentially only a list of all pieces of information the user needs to include in the database, organized in ways that will make it easy to store and retrieve that information.

Collecting all the Pieces

The first step in building a data model is to create a list of all of the elements to be included in that model. Although the list acquired during the planning discussion is a good start, it is probably not going to be complete. We, therefore, need to find other ways to gather the information. The elements include:

- lists collected during interviews.
- keywords in the mission statement and project objectives sections.
- parameters and output defined in existing reports.
- types of desired functionalities (multimedia apps will require pointers to digital objects, for example).
- capacity for interoperability with other systems or metadata standards (such as EAD, Dublin Core, OAI, or VRA Core).
- fields implied by certain types of information (such as Web site information, which would logically require a name and a URL).
- developer's intuition and experience from other similar projects.

With this preliminary list, we return to the users and obtain further information on each of these entities. We ask them to elaborate on the three entities we have extracted, again noting any additional items in our list, and asking what other information might be needed. After some discussion, we might come up with a list that would include the following:

- *help page*—to help users utilize the resource.
- *LCSH* (Library of Congress Subject Heading)—allowing interoperability with the library's online catalogue.
- *subject scope note*—describing the subject heading
- *content-type scope note*—describing the content type
- *requires proxy?*—whether resource requires a proxy server for off-campus access
- *restrictions on use*—whether restrictions, other than IP-based limitations, will apply to access
- *support name*—the name of the person who will assist users in using the site
- *support e-mail*—that person's e-mail address
- *support phone number*—his or her telephone number

To find out what types of data might be useful in filtering, it is helpful to determine under which conditions we would not want an item to be displayed. Users might also want to use other characteristics on which to group resources (excluding those not a member of the group). We thus add three more fields to the list:

- *The status* enables assigning various statuses to a site (for example, active for a available site, down for a site with a problem, and trial for a resource the library is evaluating).
- *Subscription* enables creating a list of just those resources to which the library subscribes and for which off-campus access is not allowed.
- *An alphabetical list* enables creating a list of most-used titles in alphabetical order. It might also be that, as you explore deeper into the possibilities of a project, you need to change the list you have created. For example, in our project, one thing that our users have said to us is that they would like to be able to put other things into the database, such as links to books in the online catalogue. Doing this will require two steps:
 - We change the name of the group from **Web Sites** to just plain **Sites** and the name **Web site** within the first group to **name** (to avoid confusion).
 - We add the element **format** to identify the type of site (book or Web site) and, because it is a separate concept, put it in a separate concept named **Formats**.

5.4 BUILDING THE MODEL

Now that we have our basic data groupings, we need to transform them into a data model, structuring them to build an application—taking these different entities and relating them. Part of this process involves naming our fields and concepts, which we do by taking the terms we have been using and applying the naming conventions listed in our programming standards document (for example, making everything lowercase and replacing all spaces with underscores).

We first need to make sure that each entity(table) has a field (or possibly fields) that uniquely identifies it (that is, has a value that no other record in the table would have) to serve as the primary key. If such a field does not exist, we need to create one. If we need to create one, we will use arbitrary numeric keys. We do so for several reasons:

- They will not need to be changed should content within the record change because they are not based on content.
- They can be easily generated by the system.
- They are guaranteed to be unique.
- Numeric keys are faster to process than non-numeric keys.

The first point is the most important. It is generally a bad idea to allow users to change the primary key for a table for much the same reason as why one isn't permitted to change their Social Security Number. Doing so can create a multitude of data integrity issues and many developers and many database administrators have hard rules against being able to

change a primary key. However, there are advantages to using mixed approach/meaningful keys, particularly in searching and outputting information from databases.

In the case of subjects and content types, we have decided to support items having more than one of any of these values assigned to them. Conversely, those values will usually be assigned to more than one item. This many-to-many relationship dictates that we create a linking table into which we can place a foreign key field for each data table's primary key. Laying this out so that the links between the various primary and foreign keys are shown, we come up with the representation shown in figure 5.1.

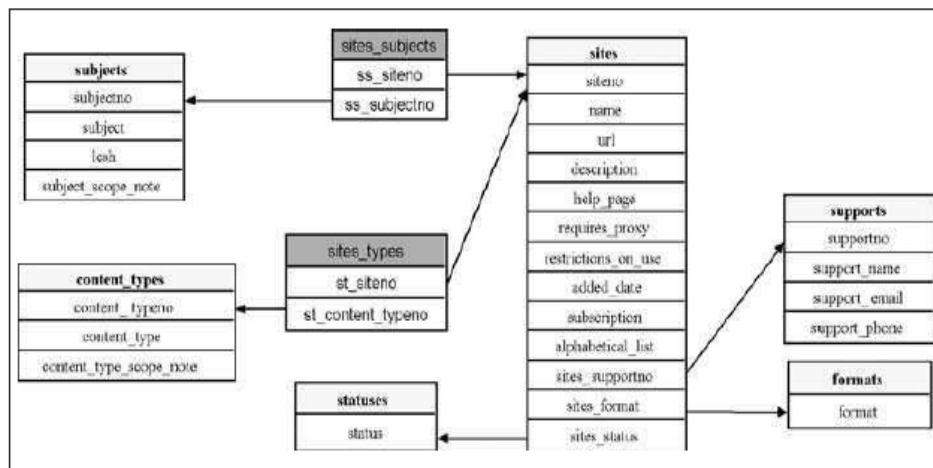


Figure 5.1

In creating linking tables, it is useful to create table names that contain the names of the tables it links with the authority table's name second (for example, sites_subjects). Not only does this let the viewer know that it is a linking table, but it also specifies which tables it links. This can be very valuable in the programming and debugging process.

Next, we make places for the foreign keys. We do so by determining what the nature of the relationship between the two tables is going to be (that is, where the foreign key will be placed) and creating the foreign key field accordingly. If it is part of a one-to-one or a one-to-many relationship, then we add an appropriate foreign key field in the foreign key table, making it the same type and size as the primary key value that will be placed there; otherwise, we create a linking table, also with fields of the same type and size as the primary key fields.

When creating the foreign key field names, we take the name of the primary key field and prepend the foreign table's name along with an underscore (for example, sites_supportno) to ensure that each field in the database has a unique name. When this would result in a

field name that is too long, I use the initials of the table followed by an underscore (for example, st_content_typedno instead of sites_types_content_typedno).

In addition to the graphic representation, you need to build a complete table and field list before moving on. As with other steps in the process, I have provided a form in which to enter this information in Grids.xls. The following outlines the process by which the **Table Definitions** grid is defined:

- Sort the **Initial Fields** grid by **Concept**, modifying each **Data Element** name to all lowercase and replacing spaces with underscores.
- Cut and paste data from the **Initial Fields** grid into the **Table Definitions** grid (in this case, columns B–J in the **Initial Fields** grid are pasted into C–K in the **Table Definitions** grid). This works because the two sets of columns have the same structure.
- Enter the lowercased name of each field group's concept into the **Table** column in the first row of each group of fields in the **Table Definitions** grid.
- Add whatever primary or secondary key fields we need to establish relations between the tables, denoting in the **Key** column whether it is a primary or foreign key. If it is a foreign key, place the table and field name of the associated primary key (in Table.FieldName format) in the **Constraints** column.
- Define the type of index (Primary, Unique, Keyword, Standard)—if any—for that field in the **Index** column.
- List what if any constraints (business rules) there are on that column in the **Constraints** column. For example, if a field is to be either yes or no, we add a constraint Y/N.
- Fill in any other column that has not been updated.
- Indicate the type of table (Data, Authority, Linking) for each concept group in the second (**Type**) column. Note that you may need to place more than one code here if a table has more than one role (e.g., a check-out record).
- Create entries for the linking tables, making sure that we make the datatype in the foreign key field the same type and size as the primary key field with which it will be linked.

Figure 5.2 shows what the **Table Definitions** grid looks like after entering our parameters.

Table	Type	Field	Length	Type	Uniq	Auth	Search	Req	Null?	Default	Limit	Key	Index	Constraints
sites	D	siteno	11	I	Y			Y	Y	N		P	P	auto_increment
		name	125	V				Y					K	
		url	125	V										
		description	125	V				Y					K	
		help_page	125	V										
		requires_proxy	1	C										Y/N
		restrictions_on_use	1	C										Y/N
		added_date	1	C										Y/N
		subscription	1	C										Y/N
		alphabetical_list	1	C										Y/N
		sites_supportno	11	I								P	S	supports.supportno
subjects	A	sites_formal	50	V		Y	Y					P	S	formals.formal
		sites_status	50	V		Y	Y					P	S	statuses.status
		subjectno	11	I	Y	Y	Y	Y	Y	N		P	P	auto_increment
		subject	100	V	Y	Y	Y	Y	Y	N				
content_types	A	lsh	100	V				Y					S	
		subject_scope_note	1	T				Y					S	
		content_hypono	11	I	Y	Y	Y	Y	Y	N		P	P	auto_increment
supports	A	content_type	100	V	Y	Y	Y	Y	Y	N		P	P	
		content_type_scope_note	1	T										
		supportno	11	I	Y	Y	Y	Y	Y	N		P	P	auto_increment
		support_name	100	V		Y	Y	Y	Y	N				
statuses	A	support_email	100	V				Y						
		support_phone	100	V				Y						
formals	A	status	50	V	Y	Y	Y	Y	Y			P	P	
		format	50	V	Y	Y	Y	Y	Y			P	P	
sites_subjects	L	ss_siteno	11	I								P	S	sites.siteno
		ss_subjectno	11	I			Y					P	S	subjects.subjectno
sites_types	L	st_siteno	11	I								P	S	sites.siteno
		st_content_hypono	11	I			Y					P	S	content_types.content_hypono

Figure 5.2

5.5 DESIGNING THE APPLICATION

Once the initial data model has been agreed upon, the next step is to design the application that will maintain and publish the data. To do this, you need to first determine the application flow. Begin by defining what if any workflow exists. If one does not exist, the developer works with the user to define one. Once you have a workflow, you proceed to map out each process that will be involved in maintaining and using the application, defining each step needed to carry out that process. You then need to define the pieces of data from your data model that need to be included within each step. You then create a user interface using mock-ups or sample screens to implement the forms that will be used to implement the queries involved in each step.

5.6 APPLICATION FLOW

Defining application flow means defining the processes involved in entering, editing, and outputting data into and from the database and defining the data involved in each step. In creating this representation, each step will use a unique view—or set of fields—in carrying out its “mission.” It is not necessary to define what fields will be associated with each view at this point. That will be done in the next step.

5.7 ADDING A RECORD

The first pair, used in adding records to the database, involves the following steps:

- When an **Add Form** page is loaded (1), it sends a set of queries to the database asking for authority lists that it can make into option lists, checkboxes, and so on. The results of these queries are then embedded into the **Add form** page.
- The server does each of the requested searches and returns the results, which are then used to create authority lists so that the user can use them in inputting data.
- The user fills out the various fields in the form and clicks on **Submit**, causing the user input to be passed to the **Insert action page** by either GET or POST (2).
- The **Insert action page** then constructs the appropriate INSERT SQL queries and sends them to the database to add the record.
- The result of the query is passed back and a response page, based on the success of the queries, is created and sent to the user.

5.8 EDITING A RECORD

Although similar in some ways to adding a record, the process is somewhat more elaborate in that the user needs to be provided with a way of selecting records:

- A **Query form** is created to allow user input (3), obtaining authority table information for authority lists as needed.
- The user fills out the page and clicks **Submit**, which sends search parameters to a **Search action page** using either GET or POST (4).
- The **Search action page** takes the parameters passed to it from the **Query form**, constructs a query (or queries), and sends the query or queries to the database.
- The database executes the queries and returns the results, which are then formatted. This formatting will include a hyperlink—using each record's primary key—to an **Editing form**, where the user will be able to modify the record's content.
- The user clicks on this link, which loads the **Editing form** (5).
- The **Editing form** retrieves the same authority lists as the **Add form**, but also the requested record and associated (linked table) information and displays the current state of the data on the update screen.
- The user makes the appropriate changes in the record and submits the changes.
- The changes are passed to an **Update action page** (6) that creates the appropriate queries and sends them to the database.
- The database executes the queries and returns the result.
- An output page with the result is presented to the user.

Note that these processes are usually carried out on the same set of data. It, therefore, makes sense for us to treat them as a unit (view).

5.9 DELETING A RECORD

On the other hand, users can also be given a link at the top of the editing page (or on the **Search action page**, for that matter) that allows them to delete the record. If they click on

that link, they can be taken to a page where they can verify their decision. Here they are given two alternatives. Those choosing to delete the record are sent to a **Delete form** (DF) to verify that the record should be deleted. If the answer is Yes, the **Delete action page** is run (DA) where the record is deleted and corresponding linked records are appropriately updated or deleted, depending on the type of links involved. If not, several options are available, such as returning to the search result or the home page.

5.10 VIEWS—BRINGING DATA AND APPLICATION TOGETHER

Now that we have decided on the views we want; we need to define the content. This means looking at each view and deciding what fields need to be included in each step of the view and how each field participates in that step. To make this easier, I have included three more forms in Grids.xls that you can use to define your views:

- *Views* define the fields used in each view and the tables into which the data will go. Their primary use is in creating data entry forms and detailing how the data in those forms are to be handled. They also allow you to specify what fields are used in each task and how those fields participate,
- *Queries* define the queries to be used in each view, particularly those that create authority record selection lists.
- *Links* define the names of tables and fields used in implementing many-to-many links within the database, to be used in adding, querying, and updating multi-table views.

5.11 DEFINING THE VIEWS

The first thing to do is to enter the names of the various views to be included in this application in the leftmost column in the **Views** grid, maintaining a row for each field to be added to the view and a blank row between the views. (If you need to add a row, you just need to right-click on a row number on the left edge of the Excel screen and select **Insert**. Another row will be inserted.) Let's begin with the **Subjects** view—the view that will be used in maintaining the subject's authority list.

1. We begin by placing the name of the view—**Subjects**—in the first column, the first row of the section. This defines the beginning of the **Subjects** view.
2. Next, we cut and paste the names of the appropriate fields from the **table Definitions** grid's **Field** column that we wish to include in this view and place them in the **Name** column.
3. We next do the same with contents from the **Table Definition** grid's **Length** column (that defines the number of characters in the field), pasting it into the **Max** column in the **Views** grid. Because the size of the column is the largest input we should allow in the form, this helps ensure that, as we build our form, we won't make any input box bigger than the capacity of the field into which the data will be going.

4. Next, we define the input types for each of the fields, using the codes. Because the first field in the list—subject—is to be assigned by the system, we leave the **Vtype** (input type) column blank (because we aren't creating an input element for it) and place the field name inside parentheses (to indicate that we are not creating an input box in the form) give it a variable type (**Vtype**) of INT (for integer) enter the name of the table into which the value will be placed into the **Table** column place (AUTO) in the (**Value**) column, to indicate the value will be automatically generated place auto_increment in the **Notes** column to indicate how it is being generated.
5. Next, we go to the second line and enter the data for the next field(subject), giving it an **Vtype** of T (text), and **Vtype** of STR (string).

Because we won't know the value for the **Size** column until we do the screen design, we skip it for now. We finish by placing the name of the table into which it will be inserted in the **Table** column. We then fill in the rest of the data for this table into the **Views** grid. Because scope_note is to be a field with textarea type, we use an **Itype** of TA (for text area). Again, we skip the **Rows** and **Cols** until we do the interface design. Once we have finished with **Subjects**, we proceed to do the same for the rest of the authority tables. Figure 5-3 shows us what the **Views** grid should look like at this point.

View	Action	Type	Auth	Name	Vtype	Size	Max	Rows	Cols	Add	Query	Display	Edit	Link	Table	(Value)	Notes
Subjects	M			(subjectno)	INT										subjects	(AUTO)	auto_increment
	M	T		subject	STR	50	100								subject		
	M	T		ish	STR	50	100								subjects		
	M	TA		subject_scope_note	STR										subjects		
Content_type	M			(content_type)	INT										content_types	(AUTO)	auto_increment
	M	T		content_type	STR	50	100								content_type		
	M	TA		content_type_scope_note	STR										content_types		
Support	M			(supportno)	INT										support	(AUTO)	auto_increment
	M	T		support_name	STR	50	100								support		
	M	T		support_email	STR	50	100								support		
	M	T		support_phone	STR	12	12								support		
Statuses	M	T		status	STR	50	50								statuses		
Formets	M	T		format	STR	50	50								formets		

Figure 5.3

Now we will look at the **Sites** view. This view is a bit more complex for several reasons: it uses authority lists from other tables, includes a wider variety of variable and input types, and updates multiple tables. We begin by entering all the fields from sites that will be used in this view, filling in the appropriate values for **View**, **Name**, **Vtype**, **Max**, and **Table** columns. In three fields—requires_proxy, subscription, and alphabetical_list—the information we obtained from our initial interviews was that values should either be yes or no. We, therefore, define them as a Y/N **Vtype** and define **Size** or **Max** values of 1 for

them. In creating the added_datefield, we will be using a different generated value—one using \$today as a variable inside the application, its value is set (as described in the **Notes** column) with PHP's date () function.

Next, we need to deal with those fields that use authority tables for inputting. First, for each field that will use an authority list for input, we create a four- to six-letter code and place it in the **Auth** column. This code provides a pointer to the **Queries** grid, where the parameters for the search to create the list will be defined. We begin with **Subject**, assigning it the code of subj, and entering it into the **Auth** column. We then do the same for **Content_Type** (), **Support** (qsupt), **Status** (qstat), and **Format** (qfmt).

We then need to define the type of HTML input tag (**Itype**) to be used in our forms in this view. In the case of the three fields where the foreign key is placed directly into a corresponding field in the sites table (sites_format, sites_supportno, and sites_status), we need to choose an input type that allows you to select only one value from the authority list. Because we can choose either select list (S) or radio buttons (R), we decide to use the select list for each.

In the case of subjects and content_types, we are setting many-to-many relationships. We,, therefore, need to take several steps. First, to allow the user to select multiple values, we need to set the **Itype** either to C (for check boxes) or M (for combo boxes). For our sample application, we will choose C for checkboxes. Second, because the input from the form could contain multiple values, we need to make the variable type one that can handle more than one value. Recalling from the previous unit that we use arrays to handle such data, we make the **Vtype** ARY. Next, so that PHP can handle the values as an array, we need to place square brackets ([]) after the field name (for example, subjects[]) so that PHP will handle it as an array and not a single-value variable. Finally, because this information is stored in another table, we put the name of that table into the **Table** column. Once we have completed our work, the **Views** grid should look like figure 5.4.

View	Action	Itype	Auth	Name	Vtype	Size	Max	Rows	Cols	Add	Query	Display	Est	Link	Table	(Value)	Notes
Subjects	M	T	qsupt	subjectno	INT			100							subjects	(AUTO)	auto_increment
	M	T		subject	STR	50		100							subjects		
	M	T		isubj	STR	50		100							subjects		
	M	TA		subject_scope_note	STR										subjects		
Content_Type	M		qctyp	content_type	INT			100							content_types	(AUTO)	auto_increment
	M	T		content_type	STR	50		100							content_types		
	M	T		content_type	STR	50		100							content_types		
	M	TA		content_type_scope_note	STR										content_types		
Support	M		qsupt	supportno	INT										support	(AUTO)	auto_increment
	M	T		support_name	STR	50		100							support		
	M	T		support_email	STR	50		100							support		
	M	T		support_phone	STR	12		12							support		
Statuses	M	T	qstat	status	STR	50		50							statuses		
Formats	M	T	qfmt	format	STR	50		50							formats		
Sites	M			siteid	INT										sites	(AUTO)	auto_increment
	M	T		name	STR	50		125							sites		
	M	T		url	STR	50		125							sites		
	M	TA		description	STR	50		125							sites		
	M	T		requires_priv	STR	50		125							sites		
	M	T		restrictions_on_use	STR	50		125							sites		
	M	TA		added_date	DATE										sites		
	M	T		subscription	INT	1		1							sites		
	M	T		subscription_note	STR	50		125							sites		
	M	T		subscription_note	STR	50		125							sites		
Sites	M	S	qfmt	sites_format	STR	50		50							sites		
	M	S	qstat	sites_status	STR	50		50							sites		
	M	S	qsupt	sites_supportno	INT										sites		
	M	C	qctyp	sites_content_type	STR	50		100							sites		
	M	C	qstat	sites_status	STR	50		50							sites		
	M	C	qsupt	sites_supportno	INT										sites		
Views	M	C	qctyp	content_type	STR	50		100							sites		
	M	C	qsupt	supportno	INT										sites		

Figure 5.4

We now need to define the appropriate query parameters in the **Queries** grid for the authority lists for which we created codes in the **Views** grid. For each code you entered into the latter, you define a line in the **Queries** grid, filling in the following columns:

View. The view in which this query will be used.

Act. The action for which this query is being used. In this case, we enter Q (for query) for each of the queries.

Auth. The four- to six-letter code entered the **Views** grid. For example, we enter **qsubj** into the **Auth** column for the subject table query.

Source: Fields. The fields to be used in the search. Because it will be filling in the value=attribute element in our authority inputting tag, the first element must be the primary key of the authority table.

This is because this is the value that will be placed in the appropriate foreign key field of the linked table. If we used a descriptive primary key, we need to place only one field name here. However, if we used an arbitrary primary key (such as auto_number), we should place the primary key first and then, after a comma, add the name of a field in that table that will meaningfully describe each record (which will identify the contents of the record to the user). In this case, because we are using an arbitrary key for subjects, we use two fields: subject no and subject.

Source: Table. Here we enter the name of the table to be searched, in this case, subjects.

Source: Where. In case we want a subset of all the records in the table, we could put the appropriate filtering WHERE clause here. However, because we don't, leave it blank.

Source: Order. To help the user, we enter the name of the field to sort on, in this case, the subject. We then proceed to do the same for the other authority table queries. The resulting entries in the **Queries** grid now look like figure 5.5.

View	Act	Auth	Source: Fields	Source: Table	Source: Where	Source: Order	Notes
Authority	Q	qsubj	subjectno,subject	subjects		subject	
	Q	qctyp	content_typeno,content_type	content_types		content_type	
	Q	qsupt	supportno,support_name	supports		support_name	
	Q	qstat	status	statuses		status	
	Q	qfmt	format	formats		format	

Figure 5.5

5.12 DEFINING TASKS

Now that we have established each of the main views, we need to define which fields are used in which task. There are three main categories of tasks: adding and inserting, querying and searching, and editing and updating (deleting, though conceptually separate, is often integrated into the editing and updating). We now go through each of these three tasks to indicate how they should be handled (see figure 5.4).

ADDING AND INSERTING RECORDS. The first task we work on is adding records to the database. We do so by defining how each field is handled in the adding and inserting process. There are five possible values, not including leaving the field blank:

X means that the value is added to the field name indicated in the **Name** column in the table named in the **Table** column. If it is to be added to a field with a different name, that name will be indicated in the **Notes** cell.

G indicates that it is generated, the type being indicated in the **(Value)** column. If `auto_increment`, we place **(AUTO)** there to indicate that it is automatically created and specify it as `auto_increment` in the **Notes** column.

H indicates a hidden value recorded in the **(Value)** column will be entered in the table indicated in the **Table** column. A logged-in user's username is one example, which might be stored in a hidden field and then entered into the appropriate field of the appropriate table when the record is added to the database.

(H) indicates that the value is a hidden one that, because it is within parentheses, is not to be added to the database. A good example of this is seen during the editing function when the primary key for the record being edited is passed to the updating action page as a hidden value so the system will know which record to update.

(D) indicates that the value is for display only and will not be recorded permanently in the database.

Blank means that the value is not part of the adding and inserting process.

In entering these values, you will also need to include values for **Size** for text inputting and **Cols** and **Rows** for text area boxes. In creating these definitions, you may find that creating mock-ups of inputting and outputting screens is a useful exercise.

Finally, because we are dealing with two fields here in the **Sites** view—subject and content_type—whose values reside in other tables, we need to define how our application can get to those fields. We, therefore, define the relational paths in our database that establish the connections to obtain that information in the **Links** grid. After placing an X

in the **Link** column of the **Views** grid (to let us know we need to look elsewhere for linking information), we go to the **Links** grid and input the following information:

View—the name of the view with which this query is to be associated **Linking Table**—the name of the table into which the link information will be placed.

Primary Table.Key—Table, the field containing the primary key for the primary data table.

PFKey Field—(primary/foreign key) name of the field in the **Linking Table** into which the **Primary Table. Key** values will be placed.

SFKey Field—(secondary/foreign key) name of the field in the **Linking Table** into which the **Secondary Table. Key** values will be placed.

Secondary Table.Key—the name of the secondary (authority) table field containing the primary key of records containing the desired value(s).

Secondary Table.Value—the name of the field in the secondary table containing the desired values

Secondary Array—the name of the array coming from the form containing an array of **SFKey Field** values **Field(=Value)**—column into which you can put other values that appear in linking tables (such as due dates in checkout records). See the result in figure 5.6.

View	Linking Table	Primary Table Key	PFKey Field	SFKey Field	Secondary Table Key	Secondary Table Value	Secondary Array	Field(=Value)
Site	sites_subjects	site\$site	ss_site	ss_subjectno	subjects.subjectno	subjects.subject	{subject_no}	
	sites_types	site\$site	st_site	content_type	content_type\$content_type	content_type\$content_type	{content_type_no}	

Figure 5.6.

EDITING RECORDS. Editing databases generally involves two tasks: selecting a record to edit and updating and saving changes. In general, each task involves its pair of form and action pages.

Querying and searching. For querying, we have two columns in the grid. In the first—**Query**—column, we define how a field is used, and if it is used at all. There are five possible values in this column:

X indicates that the field is present on the query page, using the same input values (**Itype**, **Size**, and so on) as in the view definition.

A uses an authority query to select an individual record.

M uses the field, but not as defined in the view (the character of the input being defined in the **Notes** column).

K means using a keyword search.

Blank means not included in searching.

The other column (**Display**) indicates whether the field should be displayed in the search output.

There are four codes for this column:

S means to display in a short (summary) record.

L means to display in a long (full) record.

B means to display in both.

Blank means to not display in either.

Editing and updating. Once a record is retrieved, we need to define how it participates in the editing process. As with the previous tasks, certain codes are used in this column:

X can be edited and the results are saved to the field listed in the **Name** column in the table defined in the **Table** column.

H is included in the form as a hidden variable, along with a value attribute—given in the (**Value**)—to be added to the database in that field.

(*H*) is present as a hidden variable, but is not updated (usually the primary key to allow the action form to know which record to update).

(*D*) is displayed in the editing screen but not updated in the database.

Blank is neither displayed nor updated.

5.13 INTERFACE DESIGN—CREATING THE SCREENS

While working through the **Views** grid, you will need to begin defining what fields will be used and what the interface will look like. Specifically, you need to work through how you will lay out the various inputting boxes and lists on your page. As noted, creating mock-ups is a useful exercise. There are several ways in which these can be created, including pencil and paper drawings, prototyping software, and HTML forms. Of these, creating HTML forms is probably the easiest and the most useful. If you choose to go this route, you'll want to bear several things in mind:

Using an editor that supports macros allows you to create the forms much more easily.

Naming each input element with the name of the field into which the data will eventually be placed simplifies the development process.

Creating the forms as part of the design process moves you that much farther along the road to implementation.

Creating just one form as a kind of application prototype (say, the Add form) and showing it to users for their comments allows you to make the inevitable layout and content changes to just one page. In this stage, you should attempt to create representative forms for each type of view (authority, searching, outputting, creating dynamic pages, and so on). Not only does this involve the users in the design of the interface they will be using, but it also

allows you to present the project in a manner that allows them to see how the final application will work. It allows them to provide feedback on what works and does not work and permits them to add features and fields. Because the cost of making changes goes up dramatically at each stage of a project cycle, this process can cut down significantly on expenditures, misspent time, and frustration levels.

5.14 PUTTING IT TOGETHER—PUBLIC INTERFACES

We need to define the public-access components of the application as well. We begin by creating a view name for each interface we are going to create. The planning document itemizes requests for dynamic Web pages containing links to sites by subject, organized by type of information they provide, and for searching based on subject, type of site, or descriptions of the site.

Pages by Subject

The first view we consider is for outputting subject resources by content type. The idea behind this application is that it will be written in such a way that the library Webmaster can place a link into the library's Web site that will launch an action page that takes the input search term and uses it to search and output all resources catalogued with that term.

Public Searching Interface

The other view we look at will be to support an end-user searching application. Given that this program is essentially the same task as the query function within the **Sites** view, the definition of this view looks remarkably similar, as can be seen from the **Views** grid. The only difference is that we won't be creating links to allow editing (though we will be providing a similar link to allow the display of a long record).

Testing Procedures

A formalized testing procedure must be agreed upon before beginning a project. In designing the system, it is important to create a testing document, based on the **Views** grid definitions for each task, that ensures that all data are properly handled. Such a document ideally consists of a list of all views in the application, and within each view, a list of each field that makes up that view provides a checklist for each way in which data are used in the application.

5.15 FORMALIZING THE PROCESS

Once the complete design and data specifications have been agreed upon, it is important to put them into writing. There are several reasons why this is useful:

- It provides a contractual basis for the development process. In case of any disagreements, the documentation provides the answer.
- It greatly facilitates the development process.

- It serves as documentation for the application for support by others.
- Certain documents will ideally be included in this collection and updated as things change.
- Below is a list describing the most important of these:
- *planning documents*—the initial documentation that lays out the scope, rationale, goal, and timeframe of the project
- *table definitions*—a complete list of all tables, the fields they contain, and their function within the application.
- *relational diagram*—a graphic representation of the relationships between the various tables, showing how the tables are linked between primary and foreign keys.
- *program flow*—a layout of the various processes of the programs, showing the views involved in each.
- *views grid*—definition of the tables, views, and queries involved in each point in the program flow.
- *queries grid*—a list of all queries used within each view
- *links grid*—a list of all many-to-many links maintained in each view.
- *file definitions grid*—a listing of all files used in the application, the view or views they support, and their function within the application
- *testing documents*—documents and checklists of things to test to ensure that the program is working properly.

We are now ready to implement the system. Undertaking this structured approach to planning makes the process of implementation straightforward and helps you to avoid most (though probably not all) nasty surprises with which you might otherwise have to deal.

5.16 SELF-ASSESSMENT QUESTIONS

- Q.1 Explain with examples the Project Design and its various components.
- Q.2 How to design the Project Application? Discuss with examples.
- Q.3 Describe with examples the different Views and Tasks of a Project Application.
- Q.4 Define Interface Design and Public Searching Interface with relevant examples.
- Q.5 How to bring Data and Application together? Also, elaborate on its various steps with examples.

5.17 ACTIVITY

- Plan a complete ‘Project Design’ for an academic library. Include all essential components in the Project Design which are discussed in Unit-5.

5.18 REFERENCES

Baruah, A. (2002). *Library database management*. Gyan Publishing House.

Krier, L., & Strasser, C. A. (2014). *Data management for libraries: LITA guide*. Chicago: American Library Association. Available at:
<https://books.google.com.pk/?hl=en>

Preston, C., & Lin, B. (2002). Database technology in digital libraries. *Information Services & Use*, 22(1), 9-17.

Singh, P. (2004). Library databases: Development and management. *Annals of Library and Information Studies*, 51(I), 72-81. Available at:
<http://nopr.niscair.res.in/bitstream/123456789/7488/1/ALIS%2051%282%29%2072-81.pdf>

Suseela, V. J., & Uma, V. (2017). *Data management for libraries: Understanding DBMS, RDBMS, IR technologies & tools*. Ess Ess Publications.

Westman, S. R. (2006). *Creating database-backed library web pages: Using open source tools*. Chicago: American Library Association. Available at:

[https://pdfdrive.com/download.php?title=Creating%20DatabaseBacked%20Library%20Web%20Pages:%20Using%20Open%20Source%20Tools%20\[1ed.\]0838909108,%209780838909102,%209780838998489&content=&downlurl=](https://pdfdrive.com/download.php?title=Creating%20DatabaseBacked%20Library%20Web%20Pages:%20Using%20Open%20Source%20Tools%20[1ed.]0838909108,%209780838909102,%209780838998489&content=&downlurl=)

UNIT-6

PROGRAMMING THE APPLICATION

Compiled by: **Dr Amjid Khan**

Reviewed by: **1. Dr Pervaiz Ahmad**
2. Muhammad Jawwad
3. Dr Muhammad Arif

CONTENTS

	Page #
Introduction	88
Objectives	88
6.1 Setting Up The Application	89
6.2 Implementing The Data Model.....	89
6.3. Setting Up Foreign Key Support.....	89
6.4. Creating Constraints Manually	90
6.5. Using Phpmyadmin.....	90
6.6 Creating The Configuration File	90
6.7 Programming The Application: Authority Table Maintenance.....	91
6.8 Other Maintenance Apps.....	94
6.9 Extending The Application	95
Checking For Duplicates: Fields With Duplicate Entries.....	95
Duplicate Records	96
Query Logging	97
Input Validation.....	98
Providing Useful Response	98
Output Entr.....	98
Jump To A Previous Page	98
6.10 Functions For Database Maintenance.....	98
Inputting Functions	99
6.11 Database Maintenance Functions.....	100

6.12 Testing The Application	102
6.13 Creating The Main Application.....	102
6.14 Inserting Action Page.....	104
6.15 Lost Foreign Key Values.....	111
6.16 Multiple Values Within A Field	111
6.17 Implementing Keyword Searching	113
6.18 Self-Assessment Questions	115
6.19 Activity	115
6.20 References	116

INTRODUCTION

In this unit, we will look at data and application security, creating public access applications, and good program development procedures. The steps we will follow include:

- using phpMyAdmin to implement our data model in MySQL
- creating a configuration file for the application
- building the application, including all data maintenance routines

At the end of the unit, self-assessment questions followed by practical activities are given to the students.

OBJECTIVES

After reading this unit, you will be able to learn:

- Programing and setting up the Application.
- Using phpMyAdmin to implement our data model in MySQL.
- Creating a configuration file for the Application.
- Building the Application, including all Data Maintenance routines.
- Checking for Duplicates and query logging.
- Database maintenance functions and testing of the Application.
- Implementing Keyword Searching.

6.1 SETTING UP THE APPLICATION

Before we can begin programming, we need to implement the data model in the database and create a configuration file that will contain information and parameters to be used throughout the application. We take each of these in turn.

6.2 IMPLEMENTING THE DATA MODEL

First, we implement the data model we developed during the design phase using phpMyAdmin. We do this much the same way we did in previous unit 4, by taking the **Table Definitions** grid we just created in unit 5 and using it to define the database, tables, and fields, and then setting up a user account. For the latter, we define an administrative user named Web_Sites and give it a password of ws_admin. As before, we give this user account only those rights in the database that it needs on web_sites (SELECT, INSERT, UPDATE, DELETE, and LOCK TABLES). As part of this process, there are two questions we need to ask:

Do we want to include transaction support in this application? As we discussed in unit 1, doing so greatly increases data security and integrity in multi-table apps. How do we want to maintain referential integrity (making sure all relations are properly set and maintained) between tables? Will we want to have the database maintain referential integrity for us via foreign key constraints? Will we need to program it ourselves? The difficulty is that how you answer these questions influences how you configure MySQL to store your data and thereby what you will be able to do with that data. By default, MySQL uses the MyISAM file format—one that provides neither transaction nor foreign key constraint support. To get those features, you need to set up those tables for which you wish to use these features (both the primary key and the foreign key tables) as InnoDB-type files.

However, one drawback to that course is that InnoDB does not support the FULLTEXT indexing we used in chapter 5 to implement keyword searching. For that, we need MyISAM files. Therefore, if you want keyword searching you will either need to forgo transactions and foreign key support or find another way of implementing it. I will show you that alternative (using functions in `inla_functions.php`) that will let you have your cake and eat it, too!

6.3. SETTING UP FOREIGN KEY SUPPORT

To implement foreign key constraint support within phpMyAdmin (version 2.6.2), proceed as follows in creating the database:

First, make sure that both tables (the one with the primary key and the one that will hold the related foreign keys) are the InnoDB type. To do so, you first create the tables (making sure that both the primary key and foreign key fields are of the same data type and size and that the latter is defined as NULL). Then, for each table to be included, click on its name in the left-hand frame. After the table's definition has loaded in the right-hand frame, click on the **Operations** tab in the

right-hand frame. Then go down the page to the **TableType** drop-down list and change the option to INNO DB.2Last, if you have not already done so, go into phpMyAdmin and create a primary index on the field containing the primary key and a standard index on the field that will contain the foreign key.

6.4. CREATING CONSTRAINTS MANUALLY

Next, we can create the foreign key constraints in one of two ways. The first is to click on the SQL tab inside phpMyAdmin (with websites as the active database) and enter an SQL query to create the constraint manually for each primary key/foreign key pair we want.

6.5. USING PHPMYADMIN

You can also create the constraints using phpMyAdmin's GUI interface. Although it involves a bit more work up front, it provides you with additional tools with excellent and valuable features. This includes the ability to create data dictionaries and graphical representations of your database in PDF format (including drawing lines between tables where relations exist between those tables).

6.6 CREATING THE CONFIGURATION FILE

Database applications require several items—such as database connection values—to be set on each page for the application to work. Creating a configuration file and then using PHP's `include()` function to read them into our pages will save much time and effort. At a minimum, this file should include: the connection definition parameters (host, username, password, and database name) for the user account you created, code to use those parameters to create a connection to the database, and a line, if not included in the global auto-prepend file, that will include() the PHP functions library, the code for localizing the `$_POST` super global, placing the configuration commands within a PHP block so that the PHP module will interpret the lines correctly. For this configuration information to work, we need to make sure each file in the application contains it. There are three ways to do this:

1. Use PHP's `include()` statement at the top of each page to read in the script—one possible approach, but one that requires a lot of typing, is prone to error, and needs to be changed in every file should the name of the included file change.
2. Work with the system administrator of your server to see if you can include an .htaccess file in your directory where you can type in the PHP command to automatically load it. Doing so makes it available in that directory and every directory under it.
3. Set it up so that it can be loaded via the systemwide configuration file.

6.7 PROGRAMMING THE APPLICATION: AUTHORITY TABLE MAINTENANCE

We begin by programming the authority file maintenance segment of the application (see unit 1). We do so for two reasons. First, we need populated authority tables for our dropdowns and checkboxes before we can begin programming the **Sites** view. Second, creating authority table maintenance apps is simpler and starting there will allow us to learn some basic techniques that we can build on later.

Although we could manually add records to the authority tables using phpMyAdmin, we might as well create the applications now. Besides, it is an extremely bad idea to maintain data in relational database applications through the database administration module rather than a programmed application. This is because it is far more tedious to do it that way and without the control and business rules built into a program, it can be quite easy to create data integrity problems.

SUBJECT: We start by implementing the **Subjects** view, then proceed to the other views in turn. In creating these applications, we will follow the pattern defined in unit 5, creating our adding/inserting, querying/searching, and editing/updating action page pairs in turn, plugging in the appropriate fields from the **Views** grid into each.

ADDING RECORDS: The first scripts we will create will be to add subjects' records to the database. We start with the form to input new records and name it subject_add.php. If we created a prototype inputting page in the design phase, we could use that as a model for our page. Otherwise, we take the fields and lay them out on the screen, making sure all fields from the **Views** grid are represented. In the case of subjects, we see three fields identified: subject, LCSH, and subject_scope_note.

In building this form (as well as the other forms in this application), we will be using HTML tables to define and organize the inputting elements. As we build the page, we go through the list of fields we have identified in the **Add** column of the **Views** grid with an X. We use the values we find there to decide which fields to include in the form, defining the name (using the field name), size(based on screen layout needs), and max length (based on the field size) respectively. For example, for the subject field, we write the following inputting tag:

`<input type="text" name="subject" size="50" maxlength="100">`We then go through the rest of the form in the same manner, using the text area type, rather than text, for subject_scope_note. Using the values from the **Views** grid allows us to be sure that the name of the variable will be the same as the field into which the associated value will be placed. It also helps guarantee that the max length of the inputting box will not be larger than the field into which the value will be placed. Finally, to make the form look better, we use a table within a table for the **Add Record/Clear Form** button area.

INSERTING RECORDS: The next step is to build our action page, using the name—subject_insert.php—that we defined in the action attribute of the <form> tag in the inputting form. We take the fields identified to be added using values in the **Add** column and plug them into an SQL query and send the query to the table indicated in the grid’s **Table** column. After delineating the PHP block, we create the SQL query. The syntax for the query is INSERT INTO <table> (<field_list>) VALUES(<individual_field_values>)

Although <field_list> before the VALUES keyword can be optional, it is optional only if you are inputting values for all a table’s fields in the same order as they appear in the database. If you want to insert only some fields, you must use the preliminary field list. Next, we use the PHP mysql_query() function to send the query to the database. Note that we have added or die (mysql_error()) to the end of the line. This provides some error handling and essentially says “execute the mysql_query() successfully or die (stop all execution) and print out the error message MySQL sent when it died.” Then, if the query does fail, the developer is given a message from MySQL that s/he can use to diagnose and hopefully fix the problem EDITING RECORDS Once data are in the database, we need to be able to retrieve them to edit them. We accomplish this with the subject_get.php script.

As we did in the multi_keyword_query.php file in unit 4, we create a dropdown list of subject table records from which the user can select the one to edit. Also, as we did previously, we select the primary key and display name fields from the subjects table to create the drop-down list, sorting by the display name field (subject). Then, in creating the select list in example 6.1, we use the primary key as a value and the display name to display.

Example 6.1

```

50 <select name="subjectno">
51 <?php
52     echo "<option></option>";
53     $query = "SELECT subjectno,subject FROM subjects ORDER BY subject";
54     $result = mysql_query( $query, $db ) or die( mysql_error() );
55     while ( $row = mysql_fetch_array( $result ) ) {
56         $subjectno = $row["subjectno"];
57         $subject = $row[1];
58         echo "<option value=\"{$subjectno}\">{$subject}</option>";
59     }
60     ?>
61 </select>

```

Editing screen. Because we used the table’s primary key—a subject no—as the value for each item, when the user selects a given subject and submits the form, that primary key is forwarded to the action page, subject_edit.php. There it can be used to retrieve the desired record, where that record’s contents are used to populate the values in the editing screen. The editing page is essentially the same as subject_add.php, with three important differences. The first is that the page

includes a search to retrieve the record, which it does in lines 40–42 of example 6.2, executing a search and saving the retrieved record to an associative array named \$row.

Example 6.2

```

36 <?php
37 /*****
38  * Use primary key passed from subject_get.php to retrieve record for editing
39  *****/
40 $query = "SELECT * FROM subjects WHERE subjectno=$subjectno";
41 $result = mysql_query( $query, $db ) or die( mysql_error() );
42 $row = mysql_fetch_array( $result );
43 ?>

```

Second, we use the record values in the \$row to fill in the editing screen’s value attributes which we see in example 6.3. Because, we are currently in an HTML area and we need to use the PHP engine to output the PHP variables, we use <?PHP echo \$row["<fieldname>"] ?> to output the value into the value parameter of the tag.

Example 6.3

```

60 <tr>
61 <td>
62   Subject: </td>
63 <td>
64   <input type="text" name="subject" size="60" maxlength="100" value="<?php echo $row["subject"] ?>">
65 </td>
66 </tr>
67 <tr>
68 <td>
69   LCSN:
70 </td>
71 <td>
72   <input type="text" name="lcsh" size="60" maxlength="100" value="<?php echo $row["lcsh"] ?>">
73 </td>
74 </tr>
75 <tr>
76 <td colspan="2">
77   Scope: <br><textarea cols="60" rows="3" name="subject_scope_note" wrap="virtual"><?php echo $row["subject_scope_note"] ?>
78 </textarea>
79 </td>
80 </tr>

```

The third thing we need to do we must do—is to embed the primary key name and value as a hidden variable (as noted in the **Views** grid) so we can pass it to the action page (see example 6.4). If we don’t, the update action page won’t know which record to update.

Example 6.4

```
57 <input type="hidden" name="subjectno" value="php echo $subjectno ?&gt;&gt;</pre
```

Updating action page. Once the user clicks on **Update Record**, the values are passed to the action page (subject_update.php), where an update query is created (lines 45–49 of example 6.5) and sent to the database (line 50).

Example 6.5

```
44 <?php
45 $query = "UPDATE subjects
46         SET subject='$subject',
47           lcsh='$lcsh',
48           subject_scope_note='$subject_scope_note'
49         WHERE subjectno = $subjectno";
50 $result = mysql_query( $query, $db ) or die( mysql_error() );
51 echo "Your record has been updated";
52 ?>
```

DELETING RECORDS: Deleting records follows much the same pattern as editing, with the exception that records are deleted rather than updated.

6.8 OTHER MAINTENANCE APPS

Next, we proceed to create the other data maintenance applications for the other views, one each for the **Content_Types**, **Supports**, **Formats**, and **Statuses** views. Because the first two both use arbitrary keys in the same way as subjects, we create them in the same way.

FORMATS: Formats and statuses use descriptive (as opposed to generated) primary keys and those real values are stored in the foreign key field of site records. The first point means that we need to modify our format_query.php drop-down list. Because the display value is also the primary key, we use the single value of the format field both for value and for display. However, the fact that we can change the contents of the primary key presents us with a problem: if we change the primary key value as we are editing the record, we are changing the identifier by which the action page knows which record to update. If the primary key's value is changed, the action page won't be able to specify which record to update, either in the formats or in any table where it is used as a foreign key.

As you can see in example 6.8, we get around this problem by creating a second hidden variable in the editing form (format_edit.php) in which we store the current value of the primary key (as it is in the table before editing) to PKeyVal, where it can be forwarded to the action page.

Example 6.6

```
55 <input type="hidden" name="PKeyVal" value="<?php echo $row["format"] ?>">
```

Action page: To make this change work, we need to slightly modify our updating query, and name it `format_update.php`. Finally, to maintain referential integrity with sites (remember, we didn't set up foreign key constraint support for this relation), we program the updating of foreign keys in the sites table. The query that we use for this is shown in example 6.7, with `$format` containing the new value the user has placed there and `$PKeyVal` containing the old value.

Example 6.7

```
47 $link_query = "UPDATE sites SET sites_format='$format' WHERE sites_format='$PKeyVal'";
```

STATUSES: Handling statuses is essentially the same as formats, with one exception. Because we built a foreign key constraint in the relationship between formats and sites, we don't need to create an additional query to maintain relational integrity. MySQL does it for you. Everything else is done the same way.

6.9 EXTENDING THE APPLICATION

Now that we have examined the basics of database maintenance apps, let's look at some ways to enhance our application. We will introduce you to a few additional concepts and the `ala_functions.php` library to show you how those concepts can be implemented.

Checking for Duplicates: Fields with Duplicate Entries

One thing you will want to do, especially when populating authority tables, is to prevent users from creating duplicate entries for the same value. Not only is this mandatory when creating primary keys, but it also is highly advisable when inputting the display values for authority records. Although you may not use the subject field as the primary key, you still want to make sure that you don't end up with two records with the same subject (otherwise, which one will catalogers use?). MySQL's UNIQUE index does enforce uniqueness. However, it does so by throwing out a rather cryptic (at least for most users) error message if a user attempts to enter a duplicate value. This is not the most user-friendly of approaches. It would be much better—more efficient—to check for duplicates before attempting to update the database.

There is a function—`Check_For_Dup_Fields()`—in the functions library that can handle this check for you. When calling it, you provide it with three parameters: the name of the field you wish to check, the name of the table containing the field you wish to examine, and the superglobal (`$_GET` or `$_POST`) containing the set of values passed from the form. For example, to check the subject that the user entered, you would call the function as seen in line 56 of `subject_insert2.php`, in example 6.8. The function searches the subjects table to see if it contains a record that already contains a record with the subject value the user has entered. If so, it lets the

user know that the entry would duplicate an item already in the database; the function exits before the record can be added.

Example 6.8

```
56 | Check_For_Dup_Fields( "subject","subjects", $_POST );
```

Check_For_Dup_Fields() allows you to check multiple fields in a single search. However, if you do pass it multiple fields, it will check for records where the same set of fields has the same values—as a group. Thus, if even one of the fields does not have a duplicate value, the function will not find a duplication. If you want to ensure that individual fields have unique values, you need to check each one separately.

Duplicate Records

Another type of duplication you want to avoid is creating two records with the same information (again we'll reference code from `subject_insert2.php`). One feature of the Web is that clicking on **Reload** or **Refresh** in a browser causes the server to generate the page a second time. If the page in question happens to be an action page where a record has been added to the database, reloading it will create a duplicate entry in the database for the record you just added. If you click **Reload** five times, five identical records will be added. You can avoid this by checking to see if there is already a record in the database with the current values before adding the new one. If there is, you can then warn the user and stop the execution, thus preventing the duplicate entry from being added. One of the functions in the library—`Check_For_Dup_Records()`—does exactly that. All you do is call it with the table name (`subjects`) and the superglobal containing the form input values (`$_POST`). It then searches for a record with all of those values already there (see example 6.9 from `subject_insert2.php`). If it finds one, the user is warned and the (duplicate) record won't be added.

Example 6.9

```
57 | Check_For_Dup_Records( "subjects", $_POST );
```

Both this and the previous function provide built-in responses. Although this can be useful, there may be times when you want to create your message to the user. In both cases, there is an optional additional parameter—`custom`—which, if set to `Y`, will return the number of rows retrieved by the search. You can then check to see if the number retrieved was greater than zero. If so, you can take appropriate action, such as outputting a message and exiting. On the other hand, if no record is found, you can take a different action, such as proceeding to add the record. I have provided examples of how you can do this in example 6.10.

Example 6.10

```
65 $is_there = Check_For_Dup_Fields( "subject","subjects", $_POST, "Y" );
66 if ( $is_there > 0 ) (
67     echo "Sorry - this record has already been entered";
68     exit;
69 )
70 is_there = Check_For_Dup_Records( "subjects", $_POST, "Y" );
71 if ( $is_there > 0 ) (
72     echo "Sorry - this record has already been entered";
73     exit;
74 )
```

Query Logging

It is important to maintain a transaction log of each database-modifying query that is sent to the database. Not only does it aid in debugging problems, but it also provides a means of backing up data between system backups. Creating a log involves opening a file for input, writing the query and current date to that file, and closing the file. This can require some involved programming. I have therefore provided a function in the `subject_insert2.php` script that can add this capability to our action pages. To use it in your application involves only two steps:

Create a subdirectory (folder) under the directory containing the insert/update application script, naming the directory `f` if you are working in a non-Windows environment, you will also need to make sure that the blog subdirectory is owned and writeable by the same user—usually the user `nobody`—as the Web server. Invoke the `Write Log ()` function, passing it the query name and the file name to which to add the query (see example 6.11). Invoking this function causes the contents of `$query` to be written to the `filegood.log`, located in the blog subdirectory. By consistently using the `Write Log ()` function from the very beginning of your development, you can save yourself much time and many headaches.

Example 6.11

```
90 Write_Log( $query, "good.log" );
```

If you want to use your transaction log for database recovery, use the same file for all log writing. This means that all scripts in an application that modify the database must write to the same log file. That way, the interactions can be restored in the same order in which they were initially entered. Having them in separate files will not allow for this, thereby causing potential database integrity problems (particularly when we start working with multiple tables). Although using a single subdirectory works for simple applications, things can get complicated once you begin developing more complex programs—particularly if you implement transactions.

Input Validation

Another issue concerns whether certain fields in a record are to be filled in. In some cases, such as the display fields in your authority records, you want to make sure the user inputs a value (it is not very helpful if your display name is an empty string). There are two ways to accomplish this:

- Use client-side scripting (using tools such as JavaScript) to check a designated field or fields and then, if there is no input, display a warning dialog box and not permit the user to move to the action page until the field or fields have been filled in.
- Check at the beginning of the action page before processing, but after the information has been sent, to see if there is a value in the designated fields and, if not, notify the user that a value needs to be entered in that field.

Providing Useful Response

After taking care of business, we need to provide the user with useful feedback, perhaps indicating where to go next. There are several approaches. The first is to create a simple HTML page with a link to add another record or to check the record that was just input before submitting it to the database.

Output Entry

Another approach is to let the user see what was sent to the action page. This involves going through the PHP superglobal and outputting its contents as an HTML table. Although we could do this manually, here included another function—`Show Global Vals ()`—that automates the process. In doing so, it outputs the values that were submitted to be added to the main data table.

Jump to a Previous Page

The user may wish to return to the record selection screen or search page to choose another entry to edit. You can do this by inserting a Javascript link at the bottom of the page that takes the browser back a defined number of pages. By clicking on the link below, the user would be taken back two pages (that is, over the editing screen back to the search screen):

```
<a href="javascript:history.go(-2)">Return to SearchScreen</a>
```

If you have trouble remembering exactly what the syntax for this is, there is a function that will create the above Javascript link automatically. By calling `Go Link ()` and including the number of pages you wish to go back, for example, `Go Link (-2)`, the link will be created your output page.

6.10 FUNCTIONS FOR DATABASE MAINTENANCE

You may have noticed that HTML form parameters and database queries and searches are written so that, though the individual parameters may change, the basic structure does not. We can turn this observation to our advantage by writing functions that create the structure that allows us to fill in the blanks using appropriate parameters. Not only can this make it easier and quicker to program your applications, but it also results in less typing and fewer typos (aka “bugs”). To this end, we have included functions in the `ala_functions.php` library that support a wide variety of

tasks by simply calling a function and passing it values from the grids into the appropriate parameters. Then, when the script is run, the PHP engine runs the program code and creates the HTML for you.

Inputting Functions

For example, let's take the following HTML inputting element and replace it with a function call. In this code shown in example 6.12, we see five elements: the input type (1), the name (2), the size (3), the max length (4), and the value (5).

Example 6.12

```

1      2      3      4      5
<input type="text" name="subject" size="50" maxlength="100" value="<?php echo $row['subject'] ?>">

```

We can take those elements and plug them into a TextBox() function that will create the HTML code when the page is run by the PHP engine, as seen in example 6.13.

Example 6.13

```

1      2      3      4      5
<?php TextBox("subject","50","100",$row["subject"]) ?>

```

We look at other input types, we can see a much more dramatic difference. For example, in example 6.14, we see the code for the creation of select lists we have been using.

Example 6.14

```

<td>Subject: <br>
</td>
<td> 1      2
<select name="subjectno">
<option></option>
<?php
3      4      5
$query = "SELECT subjectno,subject FROM subjects ORDER BY subject";
$result = mysql_query( $query ) or die( mysql_error() );
while ( $row = mysql_fetch_array( $result ) ) {
    $subjectno = $row["subjectno"];
    $subject = $row["subject"];
    echo "<option value=\"$subjectno\">$subject</option>";
}
?>
</select>
</td>

```


Example 6.15 shows how you can replace all of that code with a simple function call.

Example 6.15

```
<td>
Subject: <br>
</td>
<td>      1      2      3      4      5
<?php SelectList( 'subjectno', "subjectno,subject", "subjects","subject") ?>
</td>
```

Although the function might look like so much gibberish, we have numbered each parameter and shown in the previous example where that parameter is used. Specifically, we use

- SelectList—function call that will create a select list
- Subject the primary key used in the name= attribute
- Subject , subject—the two fields containing the primary key and display name values to be used in the option list items
- subjects—the table containing those fields
- subject—the field on which to sort

6.11 DATABASE MAINTENANCE FUNCTIONS

Besides items to support inputting into forms, `ala_functions.php` also includes functions to maintain the database. As with inputting, manually creating these, though not rocket science, opens the door to typos and other problems. In addition, there are several features—such as duplicate checking and logging—that you don't want to have to type in each time you create a page. (Note that using these functions will work only if you give the value name in your form the same name—including case—as the field into which the data will be entered.). Using functions allows us to simply pass the appropriate information to the functions to let them do all of the work. `Insert_Record()` is such a function. Let's see how it works.

In example 6.16, we take the code to insert a record into the subjects table and modify it to use one of the `ala_functions`.

Example 6.16

```
50 $fields = "subject, lcsh, subject_scope_note";
51 $table = "subjects";
52 $subjectno = Insert_Record( $fields, $table, $_POST, "Y", "good.log" );
53 echo "Your record has been added:<p>";
```

Although the latter has significantly fewer characters to be typed, the real advantages of using the function are more substantial:

- Several other tasks are supported by the `Insert_Record()` function, such as query logging, duplicate record checking, error checking, and support for transactions.
- The `$field` list is a simple comma-parsed list where each field name is typed in only once and is very easily read (which is important if you are dealing with large tables). If you need to make a change, you do it there, rather than having to do it both in the field list and in the `VALUES` elements (making sure that you entered both correctly and in the same order).
- MySQL's `mysql_id()` captures the `auto_increment` value of the newly created record and returns it to the calling script, where (in this case) it is saved to `$subjectno`. This value can then be used in adding linking records as needed.
- The fourth parameter can be very useful, especially if you like to see your SQL queries before running them or need to debug cranky ones.

It essentially tells the function whether to run the query. If it is `N`, the query won't run, but the function will print the query that would be run. If the parameter is `Y`, then the query will run. If it is `D`, then it will also run, but the function will not check for duplicates. There is also a `Update_Record()` function that uses similar parameters and which is available for updating data records in the database. The only difference between the two in the way that they are called is that `Update_Record()` has an additional parameter that passes the primary key to the function so it can know which record to update.

Links for editing a record: As noted, one thing that users can find useful is to be able to check—and if necessary correct—information they have input. Given that you have the primary key available to you at the end of the insert screen, you can create this link by creating a URL like that shown online 51 of example 6.17.

Example 6.17

```
49 echo "Your record has been added<p>";  
50 ?>  
51 <a href="ctype_add.php">Add another record</a>
```

Updating authority record changes: One other situation needs to be addressed—how we update authority records where the primary key has changed. We need a way to handle potentially changed primary keys. Two functions address this situation. The first, `Update_Auth_Record()`, is run on line 52 of `format_update.php` in example 6.18, using the following parameters:

`$fields`—the list of fields to be updated

`$table`—the name of the authority table

`$_POST`—the superglobal array containing the values from the form
format—the name of the primary key

`$PKeyVal`—the old value for the primary key, used to identify the record to be updated

`Y`—whether to run this function for real
good log—the name of the log file to which the query should be written.

Example 6.18

```
52 Update_Auth_Record( $fields, $table, $_POST, "format", $PKeyVal, "Y", "good.log" );  
53 Update_Auth_Links( "sites_format", "sites", $format, $PKeyVal, "Y", "good.log" );
```

Next, we need to propagate the changes to foreign key fields that link to this record. For this, we run `Update_Auth_Links()`, which we can see in line 53 of example 6.18. Here, the function is passed the name of the field containing the foreign key, the name of the table containing the foreign key field, the new value, the old value, whether the query should be run, and the log file name.

6.12 Testing the Application

Before proceeding, I would like to discuss an important part of the development process. In programming, you must make sure that your program is working correctly, that the data at any point are what they are supposed to be and are going to where they need to go. In testing, you should use the materials you developed as part of the design process (unit 5). You can use these documents to create checklists that you can use to work your way through the application, testing to make sure the program is doing what it should from beginning to end.

6.13 CREATING THE MAIN APPLICATION

Adding Records: You have now seen the basic structure of a data maintenance application using PHP and MySQL. Next, we will program the maintenance applications for the **Sites** view. We begin with adding records.

Adding Form: Looking at the **Views** grid for the **Sites** view, we note that the form for that view includes select lists, text, checkboxes, date, and yes/no.

Checkboxes: Unless you place everything in one column, creating checkbox (or radio button) input lists can be very code intensive. Each checkbox has its complete inputting tag requiring a lot of typing if there are many alternatives; even if you do use one column, it still involves a lot of typing. However, the process of automating it can be quite tricky. To produce the list in the easiest way possible—left to right, top to bottom—requires you to know how many columns you want for each row and, as you go through your results, to

know when to end one row and begin the next. It becomes even more difficult if you want to produce the list the way that many people prefer them: top to bottom within a column and then left to right with as even several rows in each column as possible.

The Check Boxes () function seeks to make this process much easier, even allowing you to select whether you want the horizontal or vertical display of your options. The snippet of code shown in example 6.19 from the site add.php form shows you how it is called. This code creates the table shown in figure 6.1.

Example 6.19

```

99 <td>
100 <td colspan="2">
101 <center>Subjects:</center>
102 </td>
103 </tr>
104 <tr>
105 <td colspan="2">
106 <?php CheckBoxes( "subjectno[]", "subjectno,subject", "subjects", "subject", "6", "1" ) ?>
107 </td>
108 </tr>
109 <tr>
110 <td colspan="2">
111 <center>Type of Content:</center>
112 </td>
113 </tr>
114 <tr>
115 <td colspan="2">
116 <?php CheckBoxes( "content_type[]", "content_type,content_type", "content_types", "content_type", "4", "1", "", "horizontal" ) ?>
117 </td>
118 </tr>

```

Although the first four parameters of the Check Boxes () function are the same as Select List (), the last two are not. The fifth (6) indicates the number of columns to include within the checkboxes table and the sixth (1) indicates the border= value for the table.

Subjects					
<input type="checkbox"/> - Agriculture	<input type="checkbox"/> - Communications	<input type="checkbox"/> - Ethnomusicology	<input type="checkbox"/> - Languages	<input type="checkbox"/> - Multidisciplinary	<input type="checkbox"/> - Social Work
<input type="checkbox"/> - Anthropology	<input type="checkbox"/> - Computer Programming	<input type="checkbox"/> - Geography	<input type="checkbox"/> - Law	<input type="checkbox"/> - Music	<input type="checkbox"/> - Sociology
<input type="checkbox"/> - Architecture	<input type="checkbox"/> - Computer Science	<input type="checkbox"/> - Geology	<input type="checkbox"/> - Leisure Studies	<input type="checkbox"/> - Musicology	<input type="checkbox"/> - Statistics
<input type="checkbox"/> - Art	<input type="checkbox"/> - Dance	<input type="checkbox"/> - History (American)	<input type="checkbox"/> - Library Science	<input type="checkbox"/> - Philosophy	<input type="checkbox"/> - Technology
<input type="checkbox"/> - Astronomy	<input type="checkbox"/> - Economics	<input type="checkbox"/> - History (European)	<input type="checkbox"/> - Literature (non-Western)	<input type="checkbox"/> - Physics	<input type="checkbox"/> - Theater
<input type="checkbox"/> - Biology	<input type="checkbox"/> - Education	<input type="checkbox"/> - History (non-Western)	<input type="checkbox"/> - Literature (Western)	<input type="checkbox"/> - Political Science	<input type="checkbox"/> - Urban Planning
<input type="checkbox"/> - Black Studies	<input type="checkbox"/> - Engineering	<input type="checkbox"/> - Human Ecology	<input type="checkbox"/> - Mathematics	<input type="checkbox"/> - Psychology	<input type="checkbox"/> - Veterinary Medicine
<input type="checkbox"/> - Business	<input type="checkbox"/> - Ethnic Studies	<input type="checkbox"/> - International Studies	<input type="checkbox"/> - Medical	<input type="checkbox"/> - Religion	<input type="checkbox"/> - Women's Studies
<input type="checkbox"/> - Chemistry					

Figure 6.1

One thing slightly different about the `CheckBoxes()` function, and its sibling `Radio Buttons()`, is that it creates a table within a table. That is, for them to function properly, they must be placed within a table, even if you are not using tablets for your input form (using a `border=0` attribute conceals the fact that a table structure is in place).

Dates: Dates can be entered in a database in any number of ways. There are not many formats, however, that permit searching, comparing, or sorting. If you don't wish to have the date be searchable, you can use any type of character-based entry format. If you do, you can opt for one of the following:

- If you are working with years only, you can use a four-character or -integer field and, as long as you are consistent in putting four characters or numbers into each field, you will be able to search and sort. If you make the field character-based, the function also allows you to use fuzzy dates. For example, in keyword searching on the field, you can truncate (201*) so that the function brings up anything in the 2001s. Also, if you consistently use four characters, you will be able to search on ranges of years, such as 2005-2015.
- You can expand on the fuzzy-date approach by making the field up to eight characters in the following format: YYYY-MM-DD. You can then truncate at any level of specificity. This is not, however, the most intuitive format to read and it is difficult to program searching against it. It is more advisable to create a data authority table containing the ranges of years that you can then use to create a dropdown box to use in data maintenance and searching.
- For the full range of possibilities in using dates, you can use MySQL's DATE field type. This allows you to sort, search, and fully compare (for example, before June 27, 2019). The only drawback is that you must provide an exact date—month, day of the month, and year.

Although you can obtain the data using a more user-friendly interface in a variety of ways, these can be code-intensive and are prone to typos and invalid date entry (February 29, 2003, for example, or September 31, 2005). What is needed is a method that allows users to enter dates in a familiar format that will also check that any date input is valid.

The Display Date() function has been designed to do just that. It creates select lists for a month, date, and year from which the user can choose.

6.14 INSERTING ACTION PAGE

In creating our action page, we need to do several things. The first is to assemble the date variables created by our `Display Date()` into a value that can be inserted into the database. We do this on line 53 of example 6.20 from site insert.php. This function takes the name of the field into which we want to place the value (added date), the three variables created by `Display Date()` (\$added year, \$added month, and \$added day), and the `$_POST` super global. From this, it builds a YYYY-MM-DD date. Then, before adding it to `$_POST`, it makes sure that the date is valid (that is, not

something like February 31). Once the function has been run, \$added date can be added to the sites table.

Example 6.20

```

53 $ _POST = Build_Date( "added_date", $added_year, $added_month, $added_day, $ _POST );
54 $fields = "name, url, description, sites_format, sites_status, requires_proxy,
55         help_page, alphabetical_list, subscription, added_date,
56         restrictions_on_use, sites_supportno";
57 $table = "sites";
58 $sitenno = Insert_Record( $fields, $table, $ _POST, $prod="Y", "good.log" );
59 if ( isset ( $subjectno ) ) (
60     Insert_Links( "sites_subjects", array( "ss_site_no"=>$sitenno ),
61         array( "ss_subject_no"=>$subjectno ), $prod="Y" );
62 )
63 if ( isset ( $content_typedno ) ) (
64     Insert_Links( "sites_types", array( "st_site_no"=>$sitenno ),
65         array( "st_content_type_no"=>$content_typedno ),
66         $prod="Y" );
67 )

```

After defining the \$fields into which we want to insert data (lines 54–56) and defining the name of the \$table containing those \$fields (line 57), we call the Insert Record () function to add the record to the database. Because we need to know what the primary key of the data record is so that we can use it for foreign key fields to link other table records to this new record, Insert Record () is called here in such a way that the value of that record's primary key returned by the function is saved to a variable named \$sitenno (line 58). We can now use the value in the Insert Links () function to create the linking table records. After making sure that there are values for \$subjectnocoming in from the form (line 59), we call the function as follows (lines 60–61).

The parameters include:

sites_subjects—the name of the table into which the links are to be placed

ss_sitenno—the name of the foreign key field in sites subjects into which the primary key of sites will be stored

\$sitenno—the name of the variable to which the primary key of the inserted sites record was saved by Insert_Record()

ss_subjectno—the name of the foreign key field in sites_subjects in which the primary key of subjects will be stored

subjectno—the name of the array, coming from site_add.php, containing the primary keys of the subjects that the user has selected (values then inserted as foreign key values in the linking records: note that square brackets are used only in input forms) *Y*—whether this is a production query (in this case, it is) *good.log*—the name of the log file to which the query should be logged.

After doing the same for the \$contentno array, we close off the page by using Display_Values() to output the data we have saved to the main data table. We provide two links: one to edit the record we just added and one to allow us to add another record.

Editing: To begin the editing process, we need to perform a search—similar to those in unit 4—to retrieve a list of records from which we choose one to edit. Because the values by which we will want to search are not in a single table, we will explore techniques to search multiple tables.

Searching: In site_query1.php, we see an example of one possible search form. It uses the same basic principles as we saw earlier. This time, however, it uses the ala_function.php library to create select lists for four of the fields (subject, content_type, format, and status) and a simple text input box for name and description. The major change comes in site_search1.php—the action page that implements the search. We begin (lines 40–49 of example 6.21) by defining some variables for the search:

Example 6.21

```
40 $fields = "sites.*";
41 $tables = "sites";
42 $link_str = "";
43 $where_str = "";
44
45 /*****
46  * Create the array of fields to display
47  *****/
48 $display_fields = array( "siteno", "name", "url", "description", "sites_format", "sites_status" );
49 $num_fields = count( $display_fields );
```

\$fields—list of fields to be retrieved (sites.*)

\$tables—all tables involved with the search (either because they contain data or are used in defining the relations), beginning with the base table, sites, and adding others as required
\$display_fields—array containing the names of the fields to be output in the order of output
\$num_fields—number of elements in the \$display_fields array (used in the for a block that outputs results)
\$link_str—in case linking statements are needed to define relations to be followed (start with it blank so that if there are no parameters it won't disrupt the eventual WHERE statement).

In example 6.21, we see the code where we go through each possible field—as in unit 4—checking to see if the user has entered a value and, if so, to add an appropriately formatted element to the \$where_ary array of WHERE conditions. When our search involves tables other than the

main sites table, we concatenate the new table names to \$tables, separating them with commas. We then add the required additional linking condition or conditions (which we obtain from the **Links** grid) to the end of the \$link_str. For example, in checking for a subject condition in lines 65–70 if \$subjectno is not blank, we add the values in lines 66–67. Also, because we are using InnoDB tables, we have resorted to the use of LIKE to search the name and description fields (line95).

Example 6.21

```

64 $x=0;
65 if ( $subjectno != "" ) {
66   $tables .= ",sites_subjects";
67   $link_str .= " AND sites.sitenno=sites_subjects.ss_sitenno ";
68   $where_ary[$x] = " ss_subjectno = $subjectno ";
69   $x++;
70 }
71
72 if ( $content_typeno != "" ) {
73   $tables .= ",sites_types";
74   $link_str .= " AND sites.sitenno=sites_types.st_sitenno ";
75   $where_ary[$x] = " st_content_typeno = $content_typeno ";
76   $x++;
77 }
78
79 if ( $sites_format != "" ) {
80   $where_ary[$x] = " sites_format = '$sites_format' ";
81   $x++;
82 }
83
84 if ( $sites_status != "" ) {
85   $where_ary[$x] = " sites_status = '$sites_status' ";
86   $x++;
87 }
88
89 if ( $name != "" ) {
90   $where_ary[$x] = " name like '%$name%' ";
91   $x++;
92 }
93
94 if ( $description != "" ) {
95   $where_ary[$x] = " description like '%$description%' ";
96   $x++;
97 }

```

We begin this section (in line 64) by setting our counter variable (\$x) to 0. Each time we add a new WHERE element, we increment (add one) to \$x. If we get to the end and \$x is still equal to 0, it probably means that no values were input—something we check in lines 115–122 in example 6.22. If it does still equal 0, we ask the user to go back and enter a value to be searched. On the other hand, if \$x is not equal to 0, it means that a value was entered, and we begin constructing our where the string (\$where_str) by stepping through the \$where_ary. We then take the \$where_str variable and, along with other constants and variables we have defined, use it to build our SQL statement.

Example 6.22

```

115 if ( $x == 0 ) {
116   echo "You need to enter a search";
117   exit;
118 } else {
119   for ( $a=0; $a<$x; $a++ ) {
120     $where_str .= " AND $where_ary[$a] ";
121   }
122 }
123
124 /*****
125  * Now let's create the $query variable, do the search (saving it to $result).
126  * Next, check to see if there are any rows in the result set. If not, let
127  * the user know that fact.
128  *****/
129 $query = "SELECT DISTINCT $fields FROM $tables WHERE 1 $link_str $where_str";
130 $records = Do_Search( $query );
131 $num_rows = count( $records );
132 if ( $num_rows == 0 ) {
133   echo "<center>No Records Found</center>";
134   exit;
135 } else {
136   echo "<center>$num_rows Records Found</center><br>";
137 }

```


In constructing this query, we have used one little SQL trick to make life easier. Because the first WHERE condition can neither have an AND in front of it (WHERE AND <condition> will give you a syntax error), nor may the last condition have a dangling AND at the end of it, and because we don't know which element will be the first condition, we have hard-coded the beginning of the conditional part of the query with were 1 (line 129 of example 6.22). In MySQL, this construct means TRUE (or "everything that could be retrieved by this query"). Although it is a bit redundant, using it means that we can place AND in front of every condition we create and do not need to figure out which one is first beforehand because 1 will always be the first condition.

- Once we have created the query (129), we pass it to a new function—Do_Search()—to do the search and return an array of records for us to process(130). Once we have that array, we use it to create the output (lines 142–169 of example 6.23):
- Create an opening <table> tag to begin our outputting table.
- Create a for loop that will walk through the \$results array, storing a new record each time to \$row.
- Use a for loop to walk through the \$display_fields array for each record from \$results, reading a name from \$display_fields into \$fld and then using that value to output the displayed string and appropriate value.
- See which field is being processed using an if statement and take appropriate actions. (If \$fld is site, for example, use it to create a hyperlink to site_edit.php, including the siteno=\$siteno as a parameter. Thus, when users click on this link, they are sent to the thesite_edit.php form, which then uses the \$siteno value to retrieve the appropriate record or records to populate an editing form. Because the \$display_fields array is processed in order, placing site no as the first element guarantees that the link will come at the top of the record.)
- Add an extra <tr><td>
</td></tr>, after a \$row is completed, to separate this record's display from the next record.
- Provide the closing </table> tag, after the loop is completed, and close off the HTML page.

Example 6.23

```

142 echo "<table border='\"0\"'>";
143 for ( $a=0; $a<num_rows; $a++ ) {
144     $row = $records[$a];
145     for ( $b=0; $b<num_fields; $b++ ) {
146         $fld = $display_fields[$b]; // save name from $display_fields array to $fld
147         $label = ucwords($fld); // make first letter of each word in $fld upper case
148         if ( $fld == "siteno" ) { // if the $fld is "siteno", create link to edit
149             echo "<tr><td align='\"right\"'><b>$label</b></td><td><a
150 href='\"site_edit.php?siteno=$row[$fld]>$row[$fld]</a> (Edit this record)</td></tr>";
151         } elseif ( $fld == "url" ) {
152             $URL = $fld;
153             $value = $row[$fld];
154             if ( trim( $value ) != "" ) {
155                 echo "<tr><td align='\"right\"' valign='\"top\"'><b>URL</b></td><td><a
156 href='\"$row[$fld]>$row[$fld]</a></td></tr>";
157             }
158         } else {
159             $value = $row[$fld];
160             if ( trim( $value ) != "" ) {
161                 echo "<tr><td align='\"right\"'
162 valign='\"top\"'><b>$label</b></td><td>$row[$fld]</td></tr>";
163             }
164         }
165     }
166     echo "<tr><td><br></td></tr>";
167 }
168 ?>
169 </table>

```

To make this searching application truly useful, we need to allow for searching multiple subjects or content types. A better alternative for keyword searching is given below.

Making Changes: After clicking on the link provided in the search output, the user is taken to the editing page (in this case, `site_edit.php`). There, the primary key (`$siteno`) is used first to retrieve the appropriate record from the sites table (using the `Get_Main_Record()` function) and then to retrieve all linking table records that have `$siteno` as the foreign key field for sites (using the `Get_Linked_Records()` function., see example 6.24).

Example 6.24

```
52 $sites_row = Get_Main_Record( "*", "sites", "siteno" );  
53 $subject_ary = Get_Linked_Records( "sites_subjects", "ss_subjectno", "ss_siteno", $siteno );  
54 $ctype_ary = Get_Linked_Records( "sites_types", "st_content_typeno", "st_siteno", $siteno );
```

In line 52 of example 6.24, the fields we want to retrieve (*), the main data table (sites), and the primary key for the main data table (**siteno**) are passed to `Get_Main_Record()` and the record is returned as an associative array back to `$sites_row`, constructing the array name using `<tablename>` as a prefix to “_row” (using prefix names for your result rows can be valuable if you ever need to deal with data from multiple tables in the same form). Once this array is created, it can be used to fill in the blanks in our editing screen. In the next two rows (53–54), we call `Get_Linked_Records()`, using information from the **Links** grid to pass it:

- the name of the linking table
- the name of the field containing the authority table’s foreign key
- the field name containing the main data record’s foreign key
- the value to search for in the main record’s foreign key field

Using this information, the function returns the results, which are then stored to the appropriate array names: `$subject_ary` and `$ctype_ary` respectively. These arrays will be used to display the current values for those multivalued authority fields in the record editing screens. Next, we create our HTML form, using `Validate ()` to require fields and creating our form appropriately. Inside the form, we place a hidden variable containing the sites table’s primary key that will be passed to the action page so it can tell MySQL which record to update.

Finally, we create the editing screen input fields using the same functions we used in `site_add.php`. However, so that the program will know what value to place into the inputting box, we add parameter to the function call—one that causes the current value for that field in the database to be placed in the value parameter in the inputting form. In the case of single-value fields where the value is coming from the main record, we use the `$sites_row` associative array, entering the field name as the array index (for example, `$sites_row["name"]` gives us the contents of the name field). As you can see in example 6.25, in the case of the linking values, we simply enter the name

of the array we created at the top of the page as the seventh parameter. For example, when creating the subjects list, we enter the name of the array of values we retrieved (in \$subject_ary).

Example 6.25



UPDATING THE DATABASE: Finally, we use the following function commands shown in example 6.26, using the data we received from the input form, to update the database.

Example 6.26

```

48 $fields = "name, uri, description, sites_format, sites_status, requires_proxy,
49         help_page, alphabetical_list, subscription, restrictions_on_use,
50         sites_supportno";
51 $table = "sites";
52 Update_Record( $fields, $table, $_POST, "siteno", "Y" );
53 if ( isset( $subjectno ) ) {
54     Update_Links( "sites_subjects", array("ss_siteno"=>$siteno),
55                 array("ss_subjectno"=>$subjectno), "subjects", $prod="Y" );
56 } else {
57     Delete_Links( "sites_subjects", array("ss_siteno"=>$siteno), $prod="Y" );
58 }
59 if ( isset( $content_typedno ) ) {
60     Update_Links( "sites_types", array("st_siteno"=>$siteno),
61                 array("st_content_typedno"=>$content_typedno),
62                 "content_types", $prod="Y" );
63 } else {
64     Delete_Links( "sites_types", array("st_siteno"=>$siteno), $prod="Y" );
65 }

```

As before, we check to make sure that there has been user input before running the Update_Links() function (example 6.26, lines 53 and 59). However, if there is not, it might mean that the user did not add any or eliminated links that had been there. To handle this situation, we assume that, in either case, the user wants no subjects or content types associated with the record. Therefore, if there was no input, we run the Delete_Links() function to delete any links that may be associated with the record. Note that in our call to Update_Links(), we have included an additional parameter. This parameter—the name of the authority table—is included so that, if the function finds an unmatched foreign key, the error message displayed on the screen will contain the table in which the primary key should have been found. This in turn helps with tracking down the problem.

6.15 LOST FOREIGN KEY VALUES

One of the problems in using authority tables for editing is that, if there is a value in a foreign key field that does not have a corresponding entry in the primary table, a value will not be displayed on the editing screen. Although this cannot occur if foreign key constraints have been established between the two tables, in some cases, you may not want to—or be able to—set those constraints, such as needing to use MyISAM file type to support FULLTEXT indexing. There are several ways in which we could end up with orphan foreign keys. For example, an authority table record is deleted without taking appropriate actions in the foreign key tables. Or if the foreign key field has had data added to it outside the application process or authority control, then there was no way to enforce the appropriate integrity. No matter how it happened, we need to have a way to deal with it. Otherwise, when the record is updated, the anomalous field entry will be lost permanently. Although it may seem rather drastic to discard data, it makes sense if you think about it. After all, the idea behind a controlled vocabulary is to control the vocabulary. Having a term in the database that is not in the authority list violates that control (to say nothing of referential integrity) and we should keep unauthorized terms out. On the other hand, the unauthorized term may be important.

We want a way to deal with it rather than just throw it out sight unseen (and not even know that it was there). To address this problem, the authority list keeps track of all default values that are passed to it and, if it finds that any of those values are orphans (don't have an associated authority table entry), a flag is set (at least one of the values to be sent to the database is set to have_f_key_problem), and an error message is displayed to the screen. In addition, it creates a hidden value named have_f_key_problem. Then, if the user attempts to update the record, each of the updating functions checks to see if any of the values coming in from the inputting form has this value. If they find it, an error message containing the primary key of the database and a statement telling the operator to contact the database administrator is displayed; and all processing stops. (If you want to allow this behavior to be overridden.)

6.16 MULTIPLE VALUES WITHIN A FIELD

Before proceeding, let's expand on some of the search techniques we are using. There will be times when you or your users will want to search by multiple values within a field (see figure 6.2 for an example).



Figure 6.2

Although we have provided ways of doing that for text input boxes, we have not done so for drop-down lists. To allow us to select multiple values, we need to change the inputting functions for subjects and content types from the `select list()` to `ComboList()` in `site_query2.php`, making sure to add square brackets (`[]`) to the end of the variable name, so that it will pass input values as an array to the action page, and defining the number of items to display—both of which we do in lines 50 and 53 of example 6.27.

Example 6.27

```

49      <td>
50          Subject: <hr></hr> ComboList( "subjectno[]" , "subjectno,subject", "subjects", "subject", "4" ) >
51      </td>
52      <td>
53          Content Type: <hr></hr> ComboList( "content_type[]" , "content_type,content_type", "content_types", "content_type", "4" ) >
54      </td>

```

Looking at line 50, we see that, after providing the label (Subject), the function is called with the following parameters (in order):

Subjectno []—the HTML name variable

subject—the fields from the **Views** grid used to create the list

subjects—the table containing the values to be placed in the list

subject—the field to sort on
4—the number of lines to show in the combo list

6.17 IMPLEMENTING KEYWORD SEARCHING

Next, we make a couple of changes to the action page (site_search2.php). First, we need to turn the array elements into a WHERE statement that can be used in an SQL query. This involves.

- 1 breaking the array into individual elements
- 2 building a WHERE element from the constituent OR parameters passed from the user (parentheses need to embrace strings with more than one value)

In the interest of saving time and effort, I have constructed a `Process_Query_Array()` that takes care of this for us. We can see an example in lines 68–76 of Example 6.28 (from site_search2.php).

Example 6.28

```
68  if ( isset( $_POST["subjectno"] ) ) {  
69      $tmp_str = Process_Query_Array( "ss_subjectno",$subjectno );  
70      if ( $tmp_str != "" ) {  
71          $tables .= ",sites_subjects";  
72          $link_str = " AND sites.sitenno=sites_subjects.ss_sitenno ";  
73          $where_ary[$x] = $tmp_str;  
74          $x++;  
75      }  
76  }
```

We check to see if a subject was input (line 68). If so, we pass the appropriate parameters (name of the field to search and name of the array containing values to be searched) to the function. The function in turn outputs a fully formatted WHERE condition, which we save to `$tmp_str`. If the string is not equal to "", something was input. We, therefore, enter the loop, where we add `site_subjects` to the `$tables` list (line 71), the linking condition needed to find sites records associated with these values added to `$link_str` (line 72), and of course, the new condition to the `$where_ary` array (line 73), incrementing `$x` in the next line (74).

As we have said, we are unable to use MySQL's FULLTEXT searching, due to our use of the InnoDB table format for the sites table. Fortunately, MySQL incorporates a very powerful

technique—regular expressions—that we can use to address the problem. Although we have nowhere nearly enough time to delve into a detailed examination of regular expressions (see the bibliography), this technique provides you with some very sophisticated tools that can look for and manipulate strings of characters within longer strings. In this case, we can use them to search for a string as a word (between spaces, punctuation, or other nonword characters) inside a field, thus avoiding the limitations we encounter with LIKE.

To provide a painless way for you to use regular expressions in your searching applications, we have created a function—`Process_Query_String()`—that creates regular expression-based MySQL queries that replicate MySQL's FULLTEXT searching capabilities. Example 6.29 (taken from `site_search2.php`) is an example. In this code, if the condition (`$description != ""`) is true, the script calls `Process_Query_String()`, passing it the field name (`description`) and the variable name (`$description`), and saves the result to `$tmp_str`. Then, in line 115, it adds `$tmp_str` to the `$where_array`. This technique allows us full Boolean searching, including by full words and phrases, with right-hand truncation using an asterisk, and both AND and OR operators (OR'ing before AND'ing after) like the `and_or_search.php` we discussed in unit 4.

Example 6.29

Example 6.29

```
113 if ($description != "") {  
114     $tmp_str = Process_Query_String("description", $description);  
115     $where_array[$x] = " $tmp_str ";  
116     $x++;  
117 }
```

At this point, we have a working Web-based application, even if it's a bit simpler than we eventually might want. We need our data to be secure, beyond the reach of both external and internal hackers.

6.18 SELF-ASSESSMENT QUESTIONS

- Q.1 Discuss programming and setting up the application with relevant examples.
- Q.2 What do you understand by the terms ‘implementing the data model’? explain with examples.
- Q.3 How would you create a configuration file for the Application? Justify your answer with examples.
- Q.4 Explain the database maintenance functions with relevant examples.
- Q.5 Write a short note on the following:
 - i. Duplicate records.
 - ii. Query logging.
 - iii. Testing the Application.
 - iv. Implementing Keyword Searching.
 - v. Inserting Action Page.
 - vi. Lost Foreign Key Values.
 - vii. Multiple Values within a Field.
 - viii. Implementing Keyword Searching.

6.19 ACTIVITY

After studying this unit carefully, design a project on ‘programing the application for an academic library’ having all those components covered/studied in this unit.

6.20 REFERENCES

Baruah, A. (2002). *Library database management*. Gyan Publishing House.

Krier, L., & Strasser, C. A. (2014). *Data management for libraries: LITA guide*. Chicago: American Library Association. Available at: <https://books.google.com.pk/?hl=en>

Preston, C., & Lin, B. (2002). Database technology in digital libraries. *Information Services & Use*, 22(1), 9-17.

Singh, P. (2004). Library databases: Development and management. *Annals of Library and Information Studies*, 51(I), 72-81. Available at:
<http://nopr.niscair.res.in/bitstream/123456789/7488/1/ALIS%2051%282%29%2072-81.pdf>

Suseela, V. J., & Uma, V. (2017). *Data management for libraries: Understanding DBMS, RDBMS, IR technologies & tools*. Ess Ess Publications.

Westman, S. R. (2006). *Creating database-backed library web pages: Using open source tools*. Chicago: American Library Association. Available at:

[https://pdfdrive.com/download.php?title=Creating%20DatabaseBacked%20Library%20Web%20Pages:%20Using%20Open%20Source%20Tools%20\[1ed.\]0838909108,%209780838909102,%209780838998489&content=&downurl=](https://pdfdrive.com/download.php?title=Creating%20DatabaseBacked%20Library%20Web%20Pages:%20Using%20Open%20Source%20Tools%20[1ed.]0838909108,%209780838909102,%209780838998489&content=&downurl=)

UNIT-7

SECURITY-RELATED TECHNIQUES

Compiled by: **Dr Amjid Khan**

Reviewed by

- 1. Dr Pervaiz Ahmad**
- 2. Muhammad Jawwad**
- 3. Dr Muhammad Arif**

CONTENTS

	Page #
Introduction	119
Objectives	119
7.1. What Does Application Security Mean?	120
7.2 Internal Threats	120
7.3 External Threats	122
7.4 Access Control	123
7.5 Using Sessions	124
7.6 Application Control	124
7.7 Logging	125
7.8 Encryption	126
7.9 Deleting Records	126
7.10 Self-Assessment Questions	127
7.11 Activity	127
7.12 References	128

INTRODUCTION

Application security is the general practice of adding features or functionality to software to prevent a range of different threats. These include denial of service attacks and other cyberattacks, and data breaches or data theft situations. This unit covers topics on application security-related techniques, internet and internet threats to the application, application access control, authorization, authentication and user sessions. It also describes application control, logging, and deleting records. At the end of the unit, self-assessment questions followed by practical activities are given to the students.

OBJECTIVES

After reading this unit, you will be able to develop:

- Internal and external threats to the application
- Application access control, authorization, authentication, and using session.
- Application control, logging, and deleting records
- Application security-related techniques

7.1. WHAT DOES APPLICATION SECURITY MEAN?

Definition: “Application security is the general practice of adding features or functionality to software to prevent a range of different threats. These include denial of service attacks and other cyberattacks, and data breaches or data theft situations.” Application security describes security measures at the application level that aim to prevent data or code within the app from being stolen or hijacked. Application security may include hardware, software, and procedures that identify or minimize security vulnerabilities. Different types of application security such as firewalls, antivirus programs, encryption programs and other devices can help to ensure that unauthorized access is prevented. Companies also can identify sensitive data assets and protect them through specific application security processes tied to these data sets. Application security is one of several levels of security that companies use to protect systems. Others include operating system security, network security and endpoint or mobile security. All these types of security are aimed at protecting clients and users of software from hacking and malicious intent.

There are several ways your application can be compromised and techniques to deal with those threats. One major step you can take is to always give your configuration files a .php extension. Although you can include files with any extension, those with .php are automatically parsed by the mod_php engine. If you use another extension, such as .inc, a crafty user could access that file and read it. Although there are tricks that you can do in httpd.conf (Apache’s configuration file) to keep that from happening, using the .php extension is the easiest approach.

7.2 INTERNAL THREATS

If you are the only person who has the right to log into your server, then these will, for the most part, not be a problem. However, if your applications will be running on a server with other Web publishers who might view your database or other application data, you start by placing that information in a file in an area outside the Web document area. You can then include () it where needed (see Setup .pdf in the companion materials download file for information on how to do so).

Another approach, and one that has the further advantage of increasing external security, is to incorporate SSL into your Apache server and place data-base maintenance applications (as opposed to public interfaces) in your HTTPS directory. Then, if you don’t provide any users access to that area, no one will be able to view or modify your files.

One drawback of using a programming language that allows you to include files from anywhere on the server is that it potentially opens your database connections to anyone. All a nefarious user needs to do is to include () your file to gain full rights to do whatever that include file allows. One way to get around that is to place code in the include file to check to see where the file that is calling it resides. If the call comes from somewhere the script has not explicitly defined (approved), the script can simply exit with a warning (or more stringent measures if you wish).

What you do is use the `$_SERVER["SCRIPT_FILENAME"]` superglobal value, which contains the name and full path—from the root directory of the Web server—of the calling script, that is, the name of the file requesting to include () your configuration file. Using PHP's regular expression function `eregi ()`, you can check at the top of your configuration file to see “who's calling”—that is, compare that information to what you've defined. If they match, the inclusion can proceed. If not, however, you can exit—perhaps logging the incident and taking appropriate measures. Examples 7.1 and 7.2 provide example code (for Windows and Unix/Linux respectively) that you can place at the top of your included file (making sure that the full path to the application directory is correct).

Example 7.1

```
38 $script = $_SERVER["SCRIPT_FILENAME"];
39 if ( !eregi("^c:/webdb/apache/htdocs/examples/chapter_7/sites", $script) ) {
40     echo "You are not allowed to use this script";
41     exit;
42 }
```

Example 7.2

```
38 $script = $_SERVER["SCRIPT_FILENAME"];
39 if ( !eregi("^/usr/local/apache/htdocs/examples/chapter_7/sites", $script) ) {
40     echo "You are not allowed to use this script";
41     exit;
42 }
```

Including such a function will cause any program that does not reside inside the designated path (`!eregi("^c:..."`) to shut down with an appropriate message before it can access any database connection or other parameters. The `eregi ()` function checks to see if two strings match: the exclamation mark (the negation symbol in almost all programming languages) in front of the function call essentially tells the computer to determine a “not match” rather than a match.

One other internal (and to some degree external) threat is allowing end-users to access password information. One way to make the users' table more secure is to encrypt user passwords. Encrypting passwords involves taking the information input into a form and, using an encryption algorithm, transforming it into something not readable by humans before storing it on the table. Then, when the user attempts to log into the application, the input is encrypted using the same method and compared to what is in the table to see if there is a match. Encryption is a one-way street: passwords cannot be decrypted from the files that store them, they can only be compared against what the user inputs, and either

verified or rejected. Casual users would thus not be able to do anything with the information in the users' table were they able to access it. Scripts that allow you to set encrypted passwords are included in the companion materials download file.

7.3 External Threats

Although hackers can compromise your system in any number of ways—and there are a host of ways to deal with it if they do. You can undertake certain precautions to make things more secure from enemies from without in the first place. Assuming that you are running PHP using mod_php rather than the CGI version, some of these include:

First, make sure that register_globals is off in php.ini (or set it to that in your local .htaccess file). If this cannot be changed, call input values (especially session values) by using their full superglobal array name (for example, \$_SESSION["name"] instead of \$name).

Always define parameters (fields, tables, and so forth) used in modifying the database and names of included files within the script, never with values input from a form.

If possible, enable SSL (Secure Sockets Layer) support on your server and place all database maintenance applications in that area. This ensures that all traffic between the user's browser (including passwords) and the server is encrypted.

If SSL is enabled, do not rely on Apache's basic_auth (use of .htaccess files with user authentication handled by htpasswd-created user files) for sensitive information. Those browser-server communications are not fully encrypted.

Set up phpMyAdmin to use cookie-based authentication on an SSL-enabled site for database administration. This allows for distributed database administration (so that people can administer their databases) and timeouts to ensure that live connections are not left open (a danger if one is doing administration on a computer where multiple people have access to the workstation).

Avoid passing user input to an external program or process such as send-mail. If you must do so, make sure to check for potential problems so that only appropriate values are passed on.

Although setting the php.ini settings for error_reporting to on and display_errors to E_ALL (every error, including undefined variables, gets reported) is fine during the development process, these settings can provide potential hackers with the information they shouldn't have. It is therefore recommended that, when placing an application in production, you set display_errors to off (to log errors to the standard Apache error log file) and error_reporting to E_ALL & ~E_NOTICE & ~E_STRICT (to keep from being inundated with error messages). Note that you can also set the error_log directive in php.ini to specify a different log file or to even e-mail you error messages that occur.

Always place query parameters within single quotes (for example, name='\$name'), even if they are numeric (neither MySQL nor PostgreSQL minds) and make sure that all query

elements have single quotes escaped, either via setting `gpc_magic_quotes` to on in `php.ini` or by using `addslashes()` on them. Not only will they make your queries work better, but it will also mean that putting something like `music'; drop DB web_sites` at the end of a search parameter will cause the nefarious code to be treated as a search parameter, and not executed as a separate SQL command.

Whenever you take the user's input and echo it to the screen, always use the `htmlspecialchars()` function to convert any code—such as Javascript functions—into HTML text

7.4 ACCESS CONTROL

Given that these applications allow for data in the database to be changed, you want to restrict access to only those persons authorized to do so. There are two levels of control:

Authentication: uses a list of those allowed into the system, checks against it when someone attempts to access the system and ensures that the person is who the person claims to be.

Authorization: verifies that the user in question is allowed to do the task he or she is attempting to perform.

Authentication: As noted, authenticating means verifying that the person is who the person claims to be. Although there are several different ways in which authentication can be handled, the easiest is to use username/password pairs. There are two techniques: The Web server's basic access control mechanisms and an application-based authentication process.

Web Server Authentication: The web server's access control does keep the curious out and allows access until the user closes the browser. Setting up server-based authentication involves two steps:

- Creating a password file (or adding to an existing one) with username/password combinations for each user.
- Configuring the Web server requires the user to enter this information.

Application authentication: We can also use our application to check users. To do so, we create a table in our database—one that we will call `users`—that we can use to authenticate our users. Although there are several useful elements we might include, we will limit ourselves here to five:

`user` (tinyint)—primary key (auto_increment) field for this table

`full_name` (varchar(50))—full name of the user

username (varchar(25))—name the user will use to log in

password (varchar(12))—password to verify their identity

rights (varchar (100))—comma-parsed list of roles (rights) the person will have in the system (we will discuss this a bit later when we address authorization).

Authorization: Although authentication does keep unwanted people out, it does not allow us to be selective in deciding who can do what in the system. To do this, we need to add an authorization layer. An authorization system assigns users access levels within the system and permissions to perform certain tasks. Then, when a user attempts to do something, rights are checked to see if the user is permitted to do so. To set up authorization, you need to define a list of words defining the roles or tasks that you want your application to support. Then, when you add a user to the user's table, you enter a comma-parsed list of the user's rights into the table's rights field. Thus, when a user attempts to perform a certain task, credentials can be checked against appropriate permissions.

7.5 USING SESSIONS

Sessions are a mechanism whereby PHP can remember individual users. The way it works is that, when a user initiates a session, the PHP engine tells the Web server to send the browser a special kind of cookie called a session ID named PHPSESSID. Once the session is set, each time the browser returns to request another page, the server checks the ID (\$_SESSION["PHPSESSID"]) in the browser to see if it is set for the server to associate the proper information with the user.

The valuable thing about sessions is that, when PHP sets up a session, it creates an associative array of values (in the form of a \$_SESSION superglobal array) that it stores in a file (in the directory defined in the session.save_path directive in php.ini), using the \$_SESSION["PHPSESSID"] as the basis for the file name. This then helps associate that file of values to the session ID stored as a cookie in the browser. Developers can use this feature to define (register) variables within the session and then use them to set values that can be accessed from any page in the application.

7.6 APPLICATION CONTROL

Controlling access to the application is one consideration. Dealing with what happens within the application once users are interacting with records in the database is another.

Transactions: When dealing with single database interactions, as we were doing in the authority maintenance apps, things were very simple: either the database was updated, or it wasn't. However, in the **Sites** view, we are dealing with multiple queries, any one of which might “go over to the dark side.” We, therefore, need to use transactions to keep

things from really being added to the database until we are sure all went well. I have included transaction support in the function libraries in the form of three functions: `Begin ()`, `Commit ()`, and `Rollback ()`. In addition, build error handling and log file maintenance into the function library. Before you use transactions, you need to have taken care of several items:

- Made sure that all tables involved in the transaction are the InnoDB type.
- Enabled sessions (for several reasons, including logging, sessions are required for the function library's transaction support to work properly).
- Called the `Begin ()` function to start the transaction process before entering any database maintenance command.
- Entered the database maintenance functions.
- Called the `Commit ()` function after the last query has run so that the results will be stored in the database (because the functions have a built-in call to `Rollback ()` if any database interaction fails, the assumption is that—if you have gotten this far—everything went well).

7.7 LOGGING

The approach to logging we discussed earlier works in a limited way. Once you begin using transactions, or if your application resides in multiple subdirectories (as it does here, using the different blog directories for the authority table maintenance and sites subdirectory for the **Sites** view), it will not work. As noted, we need to have a single log file so that all interactions to the same database can be restored in the proper order.

For this to happen properly, we need to set a global logging configuration that every script in the application can use. To set up this logging (as shown in example 7.3), you need to

- make sure that sessions are enabled in every application configuration file
- set the following variable in each configuration file:
`$_SESSION["logging"] = "Y"` (line 79 of example 7.3)
- set your logging parameters (lines 80–81)

Example 7.3

```
79 $ _SESSION["global_logging"] = "Y";  
80 $ _SESSION["global_log_path"] = "/export/www/lib/examples/Chapter_7/sites/dblog";  
81 $ _SESSION["global_log_file"] = "sql.log";
```

7.8 Encryption

Protecting our applications using authentication and authorization is a good start but does not fully protect our database. The reason for this has to do with the way that the Internet works. Left to their own devices, our Web applications will send usernames and passwords in the same way they do any other text: as plain text available for anybody with the proper software and connections to read. This, of course, can expose the information to others online who can intercept the information. In sum, they would then be able to access your database.

Just as encryption scrambles a password so that it can't be read by the "wrong" people, it can scramble all transactions between the server and the user's browser. To accomplish this, you need to implement Secure Sockets Layer (SSL) on your Web server. Any communications between the user's browser and the server are then scrambled using strong encryption methods included in SSL packages (hence their use in commercial financial transactions). Once SSL encryption is installed, you need only place your documents in the designated secure server directory (folder) and all interactions with your application will be encrypted. Please see the bibliography and Setup.pdf in the companion materials download file for more information.

7.9 DELETING RECORDS

Eliminating records from the database is a task that needs to be supported, but one that also needs to be undertaken with great care. Not only do you need to define who will be able to delete records, but you also need to determine where the information resides and what the effect of deletion will be on overall database integrity. Keep the following guidelines in mind:

- Limit the ability to delete records to highly trained and responsible individuals. One method is user authorization.
- Provide the user with a list of records that would be affected by a deletion before the record is deleted when the record the user wants to delete contains a primary key to which one or more foreign tables may be linked.
- Delete the record and take appropriate actions with all linked records once the user confirms the deletion. In some cases, the records from the linking table will be deleted. In others, if the foreign key resides within a data table, the value of the foreign key field should be set to NULL.

- Do not use phpMyAdmin or other administrative tools to delete records within a - multi-table application unless you are using built-in foreign key constraints.

7.10 SELF-ASSESSMENT QUESTIONS

- Q.1 Critically discuss the Application security-related techniques with examples.
- Q.2 What are internal and external threats to the Application? Explain with examples.
- Q.3 Write a note on Application access control, authorization, authentication, and using a session with examples.
- Q4. Explain the following:
- Application Control
 - Logging
 - Deleting records

7.11 ACTIVITY

Design a framework for Application security-related techniques with a diagram.

7.12 REFERENCES

- Baruah, A. (2002). *Library database management*. Gyan Publishing House.
- Krier, L., & Strasser, C. A. (2014). *Data management for libraries: LITA guide*. Chicago: American Library Association. Available at: <https://books.google.com.pk/?hl=en>
- Preston, C., & Lin, B. (2002). Database technology in digital libraries. *Information Services & Use*, 22(1), 9-17.
- Singh, P. (2004). Library databases: Development and management. *Annals of Library and Information Studies*, 51(I), 72-81. Available at: <http://nopr.niscair.res.in/bitstream/123456789/7488/1/ALIS%2051%282%29%2072-81.pdf>
- Suseela, V. J., & Uma, V. (2017). *Data management for libraries: Understanding DBMS, RDBMS, IR technologies & tools*. Ess Ess Publications.
- Westman, S. R. (2006). *Creating database-backed library web pages: Using open source tools*. Chicago: American Library Association. Available at: [https://pdfdrive.com/download.php?title=Creating%20DatabaseBacked%20Library%20Web%20Pages:%20Using%20Open%20Source%20Tools%20\[1ed.\]0838909108,%209780838909102,%209780838998489&content=&downlurl=](https://pdfdrive.com/download.php?title=Creating%20DatabaseBacked%20Library%20Web%20Pages:%20Using%20Open%20Source%20Tools%20[1ed.]0838909108,%209780838909102,%209780838998489&content=&downlurl=)

UNIT-8

CREATING PUBLIC INTERFACES

Compiled by: **Dr Amjid Khan**

Reviewed by : **1. Dr Pervaiz Ahmad**
2. Muhammad Jawwad
3. Dr Muhammad Arif

CONTENTS

	<i>Page #</i>
Introduction.....	131
Objectives	131
8.1 Dynamic Subject Pages.....	132
8.2. Public Searching	135
8.3 Adding Support Contact Information	135
8.4 Stopwords	136
8.5 Including Values From Foreign Tables	136
8.6 Links To Full Records	136
8.7 Paging Through Results.....	137
8.8 Setup	137
8.9 Searching.....	137
8.10 Paging Through The Results.....	139
8.11 And'ing Foreign Keys.....	139
8.12 Providing Links From Authority Entries	141
8.13 Full Keyword Searching	142
8.14 Placing Your Application In Production.....	142
8.15 Maintaining And Upgrading Your Application.....	144
8.16 Things To Watch Out For	144
8.17 Self-Assessment Questions.....	145
8.18 Activity	145
8.19 References.....	146

INTRODUCTION

Any information processed by and sent out from a computer or other electronic device is considered **output**. An example of the output is anything viewed on your computer monitor screen, such as the words you type on your keyboard. This unit looks at examples of outputting data for public use. In this unit, we will describe two applications i.e. dynamically generated subject pages and a search interface. This unit also explains in detail the process of maintaining and upgrading an application. At the end of the unit, self-assessment questions followed by practical activities are given to the students.

OBJECTIVES

After reading this unit, you will be able to:

- Dynamic subject pages
- Public searching
- Adding support contact information
- Stopwords
- Including values from foreign tables
- Links to full records
- Paging through results
- Searching AND'ing foreign keys
- Providing links from authority entries
- Full keyword searching
- Placing your application in production
- Maintaining and upgrading your application

8.1 DYNAMIC SUBJECT PAGES

Any information processed by and sent out from a computer or other electronic device is considered **output**. An example of the output is anything viewed on your computer monitor screen, such as the words you type on your keyboard. This unit looks at examples of outputting data for public use. There are two types of pages/applications i.e. dynamically generated subject pages and a search interface. In dynamic subject pages, we need to set this application up in a separate directory with a separate configuration file and separate permissions. Because public users won't need to add, edit, and delete data in the database, we adjust to accommodate the fact:

- Create a new directory—`pub_sites`—to contain our new pages.
- Create a new account—`Web_Sites_Public`—which we allow only to access the database from localhost, give no global privileges and only **SELECT** (searching) access to `web_sites`, give it a password of `pub_sites`, and remove **SELECT** permissions on the user's table.
- Create a `pub_sites.php` file containing the username, password, and database connection routines for `Web_Sites_Public` (see example 9-1) and place it in our PHP include directory (see `Setup.pdf` in the companion materials download file); create a `local_prepend.php` file in the `pub_sites` directory with a pointer to `pub_sites.php` file in it.

Example 8.1

```
45 $user = "Web_Sites_Public";
46 $password = "pub_sites";
47 $dbname = "web_sites";
48 $db = mysql_connect("localhost",$user,$password);
49 mysql_select_db( $dbname, $db );
```

To remove **SELECT** rights for users, you need to adjust in phpMyAdmin:
Log into phpMyAdmin and click on the **Privileges** link. Go to the line containing the `Web_Sites_Public` user and click on the **Edit** icon in the far-right column.

- Under **Database-specific privileges**, find the line for `web_sites` and click on **Edit**.
- Under **Table-specific privileges** either type the name of the users' table or select it from the drop-down list next to **Add privileges on the following table**.
- At the bottom of the **SELECT** column, check the **None** box and then click on the **Go** button.

Now that user will no longer have access to the users' table. In the dynamic_page.php program, I have created a basic script that outputs all of the pages within a subject, broken down by content type. The page is built in six sections:

1. We first get the subject number for a requested subject (example 8.2). If the page is called with subject=<subject>, then that value is saved to \$subject. If no value (or a bad value) was passed, we redirect the user (line 9) to a selection page (subject_select.php).

Example 8.2

```

2  /*****
3  * If $_GET["subject"] has not been set OR if Check_for_Dup_Fields() doesn't
4  * find an entry for $_GET["subject"], redirect to page to select a subject
5  *****/
6  if ( !isset( $_GET["subject"] ) || !Check_for_Dup_Fields( "subject", "subjects", $_GET, "Y" ) ) {
7      header( "Location: subject_select.php" ) ;
8  } else (
9      $subject = $_GET["subject"];
10 )

```

2. We search, once a subject has been selected, to determine the subject's primary key (subjectno) (example 8.3).

Example 8.3

```

54 $subj_query = "SELECT subjectno FROM subjects WHERE subject='$_subject'";
55 $subj_record = Do_Search( $subj_query );
56 $subj_row = $subj_record[0];
57 $subjectno = $subj_row["subjectno"];

```

3. We use the key to look up all content types represented within that subject (example 8.4).

Example 8.4

```

63 $ctype_query = "SELECT DISTINCT content_type, content_typeno
64                FROM content_types, sites_types, sites, sites_subjects
65                WHERE content_types.content_typeno = sites_types.st_content_typeno
66                AND sites_types.st_sitenno=sites.sitenno
67                AND sites_subjects.ss_sitenno=sites.sitenno
68                AND sites_subjects.ss_subjectno='$_subjectno'
69                ORDER BY content_type";
70 $ctype_records = Do_Search( $ctype_query );
71 $ctype_num_rows = count( $ctype_records );

```

4. We take the results and create an array of content types (example 9-5).

Example 8.5

```
78 $ctype=0;
79 for ( $b=0; $b<$ctype_num_rows; $b++ ) {
80     $ctype_row = $ctype_records[$b];
81     $content_type[$ctype] = $ctype_row["content_type"];
82     $content_typedno[$ctype] = $ctype_row["content_typedno"];
83     $ctype++;
84 }
```

We use the array to build a type of index at the top of the screen that will link to the output for each type (example 9-6)

Example 8.6

```
94 for ( $w=0; $w<$ctype; $w++ ) {
95     if ( $w == 0 ) {
96         echo "<a href=\"#$content_typedno[$w]\">$content_type[$w]</a> ";
97     } else {
98         echo " | <a href=\"#$content_typedno[$w]\">$content_type[$w]</a> ";
99     }
100 }
```

1. We go through the \$content_typedno array and output all records for the requested subject for each content type (example 8.7).

Example 8.7

```
110 for ( $x=0; $x<$ctype; $x++ ) {
111     $tables = " sites_types, sites, sites_subjects ";
112     $where = " sites.siteno=sites_types.st_sitenno
113         AND sites.sitenno=sites_subjects.ss_sitenno
114         AND sites_subjects.ss_subjectno=$subjectno
115         AND sites_types.st_content_typedno=$content_typedno[$x]
116         ORDER BY name";
117     $out_query = "SELECT DISTINCT * FROM $tables WHERE $where";
118     $out_records = Do_Search( $out_query );
119     $out_num_rows = count( $out_records );
120     echo "<b><a name=\"$content_typedno[$x]\">$content_type[$x]</a></b><ul>";
121     for ( $b=0; $b<$out_num_rows; $b++ ) {
122         $out_row = $out_records[$b];
123         $name = $out_row["name"]; // assign the field names to
124         $url = $out_row["url"]; // variables
125         $description = $out_row["description"];
126         echo "<li><a href=\"$url\">$name</a>"; // output a clickable URL
127         if ( trim( $description ) != "" ) { // if there is a description
128             echo " - $description"; // output it
129         }
130     }
131     echo "</ul>";
132     echo "<a href=\"#$top\">Return to top</a><p>";
133 }
```

A list of content_types is created at the top of the page (example 8.6) as it is output, each name being a link pointing to a target on the page where items of that content type will be output:

```
echo "<a href=\"#$content_typedno[$w]\">$content_type[$w]</a> ";
```

Then, as we output the records for a content type, we wrap the header in <aname=\"\$content_typedno[\$x]\"> (line 126 of example 8.7), which creates a hyperlink to the target destination. To make navigation easier, I have placed a link directing the browser back to the top of the page after each block. We can thus use the same page

for all our subjects, simply by changing the subject string passed to the script. For example, the following produces our music page:

```
<a href="http://some.library.info/subjects/dynamic_page.php?subject=Music">Music</a>
```

If we were to change the URL to read `dynamic_page.php?subject=Anthropology`, a page would be created that would output links for anthropology, and if we were to change it to `dynamic_page.php?subject=History`, a history page would be output, and so forth. Because this page requires input to know which subject to use, we need to have some way of handling those users or situations in which such a parameter is not given. As noted, if one is not, the user is directed to `subject_select.php`.

This page uses Radio Buttons (☐) to create a page that dynamically generates a list of all possible subjects for the user to select from. Thus, as subjects get added, the page will automatically accommodate the new entries.

8.2 PUBLIC SEARCHING

It is also a good idea to provide a search facility for your end users so that they can create their own result sets. To do this, we need to build a public searching application, building on techniques that we have learned, but adding some new features that make it more useful to the public. Such features include:

- support contact information,
- stopword list,
- outputting values from foreign tables associated with the site's record,
- links to full records, including information in linked tables,
- option to control the number of results per page returned
- allow Boolean **AND** searching of many-to-many fields,

full keyword searching, mimicking online search engines by enabling keyword searching, embedding phrases in double quotes, and searching multiple fields Let's go through each of these to see how we can implement them.

8.3 ADDING SUPPORT CONTACT INFORMATION

We need to provide the information we have been including in the support table to provide our public users with a way of contacting library staff to obtain help. We are storing a numeric value in the sites support no field but need to give the user the information to which it is pointing. Because the supports authority table contains e-mail addresses, one approach would be to output a mail to link for the support field.

8.4 STOP WORDS

In searching a database, one problem that users have is that they sometimes attempt to enter a natural language query—such as “Who was Christopher Columbus?”—and expect an answer-back. One way to assist those users is to create a list of “noise” words such as “who” or “was” and then filter out those words before the query is sent to the database.

8.5 INCLUDING VALUES FROM FOREIGN TABLES

When searching, not all the values that the user wants to see are contained in the main table. The user may want subjects or content types included in the search output. One approach to this is to use the primary key of the retrieved records and, following up the path of relationships, execute a search that will retrieve the associated values.

Although this function has a lot of parameters, it is straightforward and uses the following information:

- subject—the name of the field that contains the value we want to output
- subjects—the name of the authority table containing that field (foreign table)
- subjectno—the name of that table’s primary key (used to link to the linking table)
- sites_subjects—the name of the linking table
- ss_subjectno—linking table field containing the foreign key from the authority table’s primary key
- ss_sitenno—linking table field containing foreign key from main table’s primary key
\$sitenno—the primary table’s primary key to enter in the search

8.6 LINKS TO FULL RECORDS

Follow example 8.8 to create a page that outputs full records.

- Retrieves all records from data and linked tables (lines 60–62).
- Uses Auth_Vals() to construct associative arrays that can be used in outputting foreign table values (lines 63–65).
- Defines the fields to be output. Note that each array element is itself a comma-separated list. If there are two subelements, the first will be used as the label for the field in the output page and the second is the field name; if only one, then the field name will be used for both (lines 67–70).
- Creates an array to output mailto links for the supportno value (line 74), the array containing the display label, the field name, and the array of e-mail addresses.

- Calls `Display_Record()`, passing it `$fields`, main data table record, comma-parsed list of fields to make into hyperlinks, and the array to create mailto links (75).
- Calls `Display_Links()` to output linked information (lines 76–77).

Example 8.8

```

190 FOR ( $m=0; $m<Result_Count; $m++) {
191     $row = $record[$m];
192     $col = $m+0; $o=display_fields($m) {
193         $fid = $display_fields[$o];
194         $f = $row[$fid] |> "" }
195         $label = "words"
196         $label = $words($fid) // make first letter of each word in $fid upper case
197         $fitem = $row[$fid]
198         echo "<table border='1'>"
199         echo "<tr><td colspan='2' align='left'><h3>View Complete Record</h3></td></tr>";
200         $label = $fid == "url" ?
201             $URL = $fid :
202             $label = "public_id" $display_php?iten=$row[$fid] ""
203         $URL = $fid;
204         echo "<tr><td align='right'><b>URL</b></td><td><a href='\"$row[$fid]\"'>$row[$fid]</a></td></tr>";
205         $label = $fid == "email_supportno" ?
206             $supportno = $row[$fid] :
207             $label = "email" $supportno
208         if (isset($supportno)) {
209             echo "<tr><td align='right'><b>Support</b></td><td><a href='mailto:$supportno'>$supportno</a></td></tr>";
210             $label = "email" $supportno
211             $supportno = $row[$fid]
212             echo "<tr><td align='right'><b>Support</b></td><td><a href='mailto:$supportno'>$supportno</a></td></tr>";
213         } else {
214             echo "<tr><td align='right'> valign='top'><b>Label</b></td><td>$row[$fid]</td></tr>";
215         }
216     }
217 }
218
219 /*****
220 * Use Output_Link() to retrieve subjects from the subjects table, using
221 * the linkid parameter to search for.
222 * See the documentation for the Output_Link() function for details
223 * on how to use it. You have the string, you create the
224 * linkid and output the string.
225 *****/
226 $subject_rec = Output_Link("subjects", "subjectno", "subjectno");
227 echo "<table align='right'>"
228 echo "<tr><td align='right'> valign='top'><b>Subject</b></td><td>$subject_rec</td></tr>";
229 echo "</table>";
230 $displayed_count++;
231 if ($displayed_count == $num_to_display) {
232     echo "</table>";
233     echo "<br>";
234     echo "</table>";
235     echo "</table>";
236 }

```

8.7 PAGING THROUGH RESULTS

Paging through a query result is the process of returning the results of a query in smaller subsets of data, or pages. To return a page of data from a data source without using the resources to return the entire query, specify additional criteria for your query that reduce the rows returned to only those required.

This is not something that will work well if the number matched is large and/or there are many fields to be displayed. One way to handle this is to break the results down into manageable chunks and allow the user to page through them. Although you could write the retrieved records to a temporary file and write an application to go through them. Fortunately, MySQL provides a wonderful feature called LIMIT you can use to limit your output to a defined range of records.

8.8 SETUP

To make paging work, we need to use sessions to store parameters between pages, thereby allowing us to use them to redo the search with different LIMIT parameters and permitting us to page through large results sets. As we have seen with authentication, using sessions requires that the session variables be registered each time a page is loaded.

8.9 SEARCHING

Next, let's set up the search page. We begin by taking the query file (sites/site_query2.php) and saving it to a new name in a different directory (pub_sites/public_query.php). Next, we save the search file (sites/site_search2.php) similarly (as pub_sites/public_search.php). We

then proceed to modify `public_search.php` to include the appropriate code (see example 8.8 for the following example references).

Example 8.9

```

44 $SESSION["dbquery"] = "";
45 $pager = "pager.php";
46 $SESSION["pager"] = $pager;
47 $fields = "sites.*";
48 $tables = "sites";
49 $link_str = "";
50 $where_str = "";
51
52 /*****
53 * Create a $support_ary array to include support email addresses for those
54 * sites where a support person has been identified in the system.
55 *****/
56 $support_ary = Auth_Vals( "supportno","support_email","supports" );
57
58 /*****
59 * Next, we define the fields to display ($display_fields) and count the
60 * number of fields that are in the array. Then, we assign the contents
61 * of those variables to the session variable of the same name.
62 *****/
63 $display_fields = array( "siteno", "name", "url", "description", "sites_format", "sites_status", "sites_supportno" );
64 $SESSION["display_fields"] = $display_fields;
65 $display_fields_num = count( $display_fields );
66 $SESSION["display_fields_num"] = $display_fields_num;
67 $num_to_display = $SESSION["num_to_display"];

```

After initializing the `$SESSION["DB query"]` variable by setting it to "to clean out any old query string that it may contain in line 44, we define the page that will handle paging through the results (`pager.php`) and define the base `$fields`, `$tables`, `$link_str`, and `$where_str` to be used in the search (lines 45–47). Next, we store our values to the session variables that we will be using line 63–64 (`$display_fields`), and line 65–66 (`$display_fields_num`). Then, as we can see in example 8.10, we define additional variables in line 153 (`$dbquery`) and line 167 (`$result_num_rows`) as well as localizing (making a superglobal value into a local value) `$num_to_display` (line 67 of example 8.9).

Example 8.10

```

152 $query = "SELECT DISTINCT $fields FROM $tables WHERE 1 $link_str $where_str ORDER BY siteno";
153 $SESSION["dbquery"] = $query;
154 $records = Do_Search( $query );
155 $result_num_rows = count( $records );
156 /*****
157 * Next, we check to see if any records were returned by the query. If not,
158 * the user is informed of that fact and the script exits. If records are
159 * found, the number of records retrieved is displayed on the page and the
160 * records are output.
161 *****/
162 if ( $result_num_rows == 0 ) {
163     echo "No Records Found";
164     exit;
165 } else {
166     echo "<center>Your query returned $result_num_rows records</center>";
167     $SESSION["result_num_rows"] = $result_num_rows;
168 }
169
170 /*****
171 * Next, if we have retrieved more than $num_to_display number of records,
172 * we put in a navigational header. The URL for each option invokes the
173 * pager.php file, passing it the option as a GET variable (type=<str>).
174 * When the user clicks on that link, pager.php will be invoked and it will
175 * use that parameter, along with the $SESSION variables, to construct
176 * another query to retrieve $num_to_display records to display.
177 *****/
178 if ( $result_num_rows >= $num_to_display ) {
179     echo "<center><a href=\"\$pager?type=first\">First</a> | ";
180     echo "<a href=\"\$pager?type=next&starting_record=0\">Next</a> | ";
181     echo "<a href=\"\$pager?type=prev&starting_record=0\">Previous</a> | ";
182     echo "<a href=\"\$pager?type=last\">Last</a></center><br>";
183 }

```

We create navigation links to place at the top of the page to allow us to look at different sets of the retrieved records (lines 178–183 of example 8.10). Because it doesn't make sense to page through results if we retrieve fewer records than our maximum-per-page number, we only put up the navigation links if the number of retrieved records is greater than our \$num_to_display value (lines 178–183). Each of the lines invokes our \$pager file (pager.php), passing it the parameter for First, Next, Prev, and Last. In the second and third instances, we also pass it the number the current search started at (initially 0). Those parameters are then used in pager.php to define which subset of records to retrieve. Thus, though public_search.php handles the initial search, any subsequent output will be handled by pager.php.

8.10 PAGING THROUGH THE RESULTS

In creating the paging script, we begin by taking the searching and outputting sections of public_search.php and saving them to a new file named pager.php. Then, at the top of the paging script, after creating the \$support_ary array and localizing the session variables (storing them to local variables) to make them easier to use (lines 52–57 of example 8.11), we check to see with which record the users began the previous retrieval set and use that to define their next SQL query (lines 59–75). We then create our HTML page, using our updated starting numbers to redo the First/Next/Previous/Last navigation bar. We then search with our new parameters, using the same code to output the results as was used in public_search.php.

Example 8.11

```

52 $num_to_display = $_SESSION["num_to_display"];
53 $result_num_rows = $_SESSION["result_num_rows"];
54 $display_fields = $_SESSION["display_fields"];
55 $display_fields_num = $_SESSION["display_fields_num"];
56 $dbquery = $_SESSION["dbquery"];
57 $pager = $_SESSION["pager"];
58
59 if ( $type == "first" ) {
60     $start = 0;
61 } elseif ( $type == "next" ) {
62     if ( ( $starting_record + $num_to_display ) >= $result_num_rows ) {
63         $start = $starting_record;
64     } else {
65         $start = $starting_record + $num_to_display;
66     }
67 } elseif ( $type == "prev" ) {
68     if ( $starting_record - $num_to_display < 0 ) {
69         $start = 0;
70     } else {
71         $start = $starting_record - $num_to_display;
72     }
73 } elseif ( $type == "last" ) {
74     $start = $result_num_rows - 1;
75 }

```

8.11 ADDING FOREIGN KEYS

Until now, we have looked at using Boolean AND and OR between fields within a table and using OR between field value parameters in foreign key tables. You could radically de-normalize your database, store subjects as keywords in a keyword field in your main

site's table, and then use keyword searching to retrieve them, this approach creates other problems, such as how you can maintain authority tables, enforce uniform entry of values, and create drop-down lists.

The problem is that the approach you think should work does not:

- `SELECT*`
- `FROM sites, sites_subjects`
- `WHERE sites.siteno=sites_subjects.ss_siteno`
- `AND sites_subjects.ss_subjectno=3`
- `AND sites_subjects.ss_subjectno=28`

What this query is doing is requiring that the `sites_subjects` table have a single record where `ss_subjectno` is equal both to 3 and to 28 at the same time—something that obviously won't happen.

The solution is to create separate aliases (copies, if you will) of the linking table—one for each term to be searched—and then use those separate aliases in your `WHERE` statement. The result is that the following does work:

- `SELECT sites.*`
 - `FROM sites, sites_subjects AS s1, sites_subjects AS s2`
 - `WHERE s1.ss_siteno=s2.ss_siteno`
 - `AND sites.siteno=s1.ss_siteno`
 - `AND s1.ss_subjectno=3 AND s2.ss_subjectno=28`
- Breaking this down, we can see how it is constructed:

- Line 1—list of fields to retrieve
- Line 2—Use aliases (`sites_subjects` as `S1`, `sites_subjects` as `S2`) to create separate logical instances of the linking table—in this case the `sites_subjects` table—for each term to be searched
- Line 3—make sure that all the aliases' primary key (in this case `ss_siteno`) field values are linked (equal to each other)
- Line 4—make sure that one of them is equal to `sites.siteno`
- Line 5—use a separate alias of `sites_subjects` for checking each foreign key (`ss_subjectno`) field

value to see if it is equal to a value coming in from the searching form. If all of this is a little beyond what you want to deal with, don't worry. You can use the `Process_Query_And_Array()` function in the `ala_functions.php` library to do the job for you. Example 8.12 from `public_search2.php` shows one way to use it.

Example 8.12

```

92 if ( isset( $subjectno ) ) {
93     if ( isset( $Operator1 ) && $Operator1 == "AND" ) {
94         $tmp_ary = Process_Query_And_Array( "ss_subjectno", $subjectno, "sites_subjects",
95                                             "sites.sitenno", "ss_sitenno" );
96
97         if ( count( $tmp_ary ) != 0 ) {
98             $tables .= ", $tmp_ary[0]";           // define $tables the 0th (1st) element
99             $link_str .= " $tmp_ary[1] ";        // define $link_str w/2nd element
100             $where_ary[$x] = $tmp_ary[2];        // where string using the 3rd element
101             $x++;
102         }
103     } else {
104         $tmp_str = Process_Query_Array( "ss_subjectno", $subjectno );
105         if ( $tmp_str != "" ) {
106             $tables .= ", sites_subjects";
107             $link_str = " AND sites.sitenno=sites_subjects.ss_sitenno ";
108             $where_ary[$x] = $tmp_str;
109             $x++;
110         }
111     }

```

8.12 PROVIDING LINKS FROM AUTHORITY ENTRIES

Before concluding, let me show you one more technique, which we can see at work in `public_search2.php`. It involves using values (in this case subject headings) to create links that we can then use to do another search. We do this by first adding two parameters to the call to `Output_Links()` on line 275 of Example 8.13:

Y (to indicate we want to create links)

`public_search2.php` (the name of the file that will do the follow-up search).

Example 8.13

```

274 $subject_str = Output_Links( "subject", "subjects", "subjectno", "sites_subjects",
275                             "ss_subjectno", "ss_sitenno", $sitenno, "Y", "public_search2.php" );
276 echo "<tr><td align='right' valign='top'><b>Subject:</td><td>$subject_str</td></tr>";
277 echo "<tr><td><br></td></tr>";

```

Example 8.14

```

42 if ( isset( $_GET["lookup_val"] ) ) {
43     $subjectno = array( $_GET["lookup_val"] );
44 } else {
45     $subjectno = $_POST["subjectno"];
46 }

```

Next, because the link created by `Output_Links()` creates a GET variable named `$lookup_val`, we place code at the top of `public_search2.php`, which will process the page if such a variable is passed to it. We see this done in lines 42–45 of example 8.14. If it does exist, we create a one-element array called `$subjectno` and save `$_GET["lookup_val"]` as that element. Otherwise, we save `$_GET["subjectno"]` to the `$subjectno` variable. By setting this variable, the search will be done just as if the value had come from `public_query2.php`.

8.13 FULL KEYWORD SEARCHING

For such an approach to work in a fielded database, we need to be able to search through multiple fields at a time as well as able to treat collections of words within double quotes as a single phrase and those that are not as simple keywords.

To address this need, we can use another function in the `ala_functions.php` library—`Process_Quoted_String()`, which also supports right-hand truncation.

As with `Process_Query_Array()` earlier, we use this function to create where elements that we can plug into a query. We can see a simple example of how it is used in example 8.15 where we pass it only two parameters: the name of the field to be searched and the variable containing the string to be processed (for an example of how it can be used to search multiple fields, see `public_keyword_search.php`).

Note that, due to the complexities involved in doing so, this function is not able to search values located in tables linked via many-to-many joins (in this case, either `subjects` or `content_types`). It does, however, provide the ability to search for terms within any field within a single table.

Example 8.15

```
12  if ( $keywords != "" ) {
13      $tmp_str = Process_Quoted_String( "description,name", $keywords );
14      $where = " $tmp_str ";
15  } else {
16      echo "You need to enter a search";
17      exit;
18  }
```

8.14 PLACING YOUR APPLICATION IN PRODUCTION

Once the application has been written and tested, it's time for users to try it out. When presenting it, it is a good idea to walk them through the entire program, showing them how

to use it and—in the case of searching scripts—what values they can use to search the database. Users should also be given a list of tasks for which the system was designed and then be left to work through each task in this list, looking for data entry or storage problems. The process of moving an application into production involves several steps:

- Use phpMyAdmin to execute a complete backup dump of the data structures and data, exporting the results to a file.
- Clean up test data in the database.
- Create a second database dump containing both structures and data.
- Define a new production environment where the application will live and copy the complete source code tree from your development area to the new area.
- You may need to change internal paths, configuration information (such as logging directories and files), and other parameters so that the application will work in the new environment and will not conflict with the development version. If possible, set up the new production environment on a separate computer so that you can maintain the same database, configuration, and path information for each version of the database. (If you do not, moving new versions into production will be tricky and time-consuming.)
- Define a new database and use the second dump file to create and populate the tables.
- Create or modify MySQL user accounts to allow access to the new database, adding this information to your configuration files, as needed.
- Test the application fully, afterwards deleting any test data that might have been added to the database and reloading the database.
- Establish and implement a backup plan (if you have not already done so).
- Continue to use the original area for development. If the development will be substantial, create a third area for that development, maintaining this second area for bug fixes and minor feature enhancements.
- Set up Web server authentication access control for your development area or areas. Then, if you want users to test the system, you can temporarily add them to the system and delete them when testing is over. Although the development area does not pose a security threat, placing password control on the area will keep users from mistakenly continuing to use it as a production area when testing has been completed.
- Make appropriate links on your Web site to the new application.

8.15 MAINTAINING AND UPGRADING YOUR APPLICATION

Once the program has been placed into production, your work doesn't end. Bugs will show up, features will need to be enhanced or added, and users' questions will need to be addressed. You need to make sure that you are set up to handle this type of maintenance. In addition to having separate support and production areas, there are several things you should do to make this process work properly.

- The first is to designate who will be responsible for maintenance. Make sure in doing so that you have adequately trained persons available (either on staff or outsourced) who will be able to support the application down the road, both in terms of programming and database maintenance.
- The second is to develop processes for upgrading the application. Because users will find new features they would like to include in the application, you need to have a process by which such upgrades can be suggested, evaluated, and implemented. This process will also need to include procedures—like those in chapter 6—in which functional requirements are established, data modelling done, and the application design. Once the enhancements are agreed upon, you will need to implement them in your development area and fully test the resulting application.
- The third is to document your application. This is a critical part of any development project—so important in fact that I devote much of the next chapter to it.

8.16 THINGS TO WATCH OUT FOR

It is important to understand that several things can affect your application that have nothing to do with the code you have written. Changes in several things, including the `php.ini` file, Apache's configuration file, and the functions library can end up causing your application to crash. Although there is not enough time to get into all of them, there are a few rules of thumb you should consider:

- Be extremely careful of the changes you make in global function files. Changing how they work can introduce all sorts of hidden bugs into your apps. Think through all modifications you want to make to those files before making changes and test the resulting library with all applications that use the functions.
- Never make changes to the production version of the function library. Any syntax or other error you make there will cause every PHP application on your Web server to crash. A better idea is to edit it in the development environment, test all your apps with the new setup, and then propagate the changes into your production environment.

- Work with the server administrator—if you are not the one responsible for administering your Web server—so that no one changes basic configuration files without letting you know. Such changes might affect all PHP files throughout the system—something that .htaccess files can help address in multiuser systems, such as ISP-based situations.

We have now programmed our application, ensured data security, designed our user interfaces, completed testing, and placed the application in production.

8.17 SELF-ASSESSMENT QUESTIONS

Q.1 Explain the steps involved in creating a public interface with relevant examples.

Q.2 Describe the following with examples:

- Dynamic subject pages
- Public searching
- Adding support contact information
- Stopwords

Q.3 Write a short note on the following:

- Including values from foreign tables
- Links to full records VS Full keyword searching
- Paging through results
- Placing your application in production
- Maintaining and upgrading your application

8.18 Activity

Study this unit carefully and design, design a public search interface having all features of unit 08.

8.19 REFERENCES

- Baruah, A. (2002). *Library database management*. Gyan Publishing House.
- Krier, L., & Strasser, C. A. (2014). *Data management for libraries: LITA guide*. Chicago: American Library Association. Available at: <https://books.google.com.pk/?hl=en>
- Preston, C., & Lin, B. (2002). Database technology in digital libraries. *Information Services & Use*, 22(1), 9-17.
- Singh, P. (2004). Library databases: Development and management. *Annals of Library and Information Studies*, 51(I), 72-81. Available at: <http://nopr.niscair.res.in/bitstream/123456789/7488/1/ALIS%2051%282%29%2072-81.pdf>
- Suseela, V. J., & Uma, V. (2017). *Data management for libraries: Understanding DBMS, RDBMS, IR technologies & tools*. Ess Ess Publications.
- Westman, S. R. (2006). *Creating database-backed library web pages: Using open source tools*. Chicago: American Library Association. Available at: [https://pdfdrive.com/download.php?title=Creating%20DatabaseBacked%20Library%20Web%20Pages:%20Using%20Open%20Source%20Tools%20\[1ed.\]0838909108,%209780838909102,%209780838998489&content=&downurl=](https://pdfdrive.com/download.php?title=Creating%20DatabaseBacked%20Library%20Web%20Pages:%20Using%20Open%20Source%20Tools%20[1ed.]0838909108,%209780838909102,%209780838998489&content=&downurl=)

UNIT-9

DEVELOPMENT PROCEDURE

Compiled by: **Dr Amjid Khan**

Reviewed by **1. Dr Pervaiz Ahmad**
2. Muhammad Jawwad
3. Dr Muhammad Arif

CONTENTS

	<i>Page #</i>
Introduction	149
Objectives	149
9.1 Overview	150
9.2 Establishing A Process	150
9.3 Implementing Standards.....	150
9.4 Quality Assurance	150
9.5 Documentation	151
9.6 Technical.....	151
9.7 Code	151
9.8 User	152
9.9 Help Screens	152
9.10 Debugging.....	152
9.11 Self-Assessment Questions	153
9.12 Activity.....	153
9.13 References.....	154

INTRODUCTION

When undertaking any type of programming project, you want to follow certain practices while you're writing the code for your application. These are designed to standardize your approach, make code readable and—more important—supportable, and ensure proper functioning with a minimum of effort. They break down into four areas: establishing a development process, implementing programming standards, establishing quality assurance procedures, and using good debugging techniques. This unit covers all these aspects with examples. At the end of the unit, self-assessment questions followed by practical activities are given to the students.

OBJECTIVES

After studying this unit, you will be able to understand:

- Program development procedure, establishing process and implementing standards
- Quality Assurance
- Documentation (technical, code, users, help screens and debugging).

9.1 OVERVIEW

When undertaking any type of programming project, you want to follow certain practices while you're writing the code for your application. These are designed to standardize your approach, make code readable and—more important—supportable, and ensure proper functioning with a minimum of effort. They break down into four areas: establishing a development process, implementing programming standards, establishing quality assurance procedures, and using good debugging techniques.

9.2 ESTABLISHING A PROCESS

It is important to define a development process to keep the project on track and to make things work as smoothly and as efficiently as possible. Although there are several ways you can do this, such as:

- Define who is going to be given what tasks.
- Build applications one step at a time
- Test and debug as you go—it is much easier to debug ten lines of code than two hundred and fifty.
- Review code periodically, making sure that code is following programming standards (see below).
- Check into CVS or other version control system periodically.
- Test the application fully using documentation.

9.3 IMPLEMENTING STANDARDS

As with any area of automation, consistency, and clarity in how programs are laid out make them much easier to develop and support. Although it can be tempting for developers to throw themselves into the coding and not worry about being careless and/or cute in how they name things, the results of such development can be a trial to understand and support. Above all, inconsistency in naming and practices requires developers to remember a lot more than they should need to (and more than people unfamiliar with the code will want to). Although some are somewhat arbitrary, the same is true of almost any standards one might create. What is important is that standards are consistent and make sense within the organization where they are adopted, not that they are absolute truth.

9.4 QUALITY ASSURANCE

As you develop your application, submit each section to a quality-review process to make sure that it is being written correctly. Nothing is more unnerving to end-users than to have an application malfunction, leaving possibly their self-assurance (and certainly their confidence in the application) shaken. Taking the time to get it right the first time will make their (and certainly your) lives much easier.

These are two things you can do to implement successful quality assurance procedures. The first is to develop testing documents. Using the testing procedures, we have talked about can be very valuable in finding errors.

The second is to institute code reviews. In addition to debugging, reviewing code line-by-line can be extremely useful in producing quality applications.

The idea is to work with someone else—preferably an accomplished programmer—going through the programs and explaining what is (at least supposed to be) happening, what you are trying to do, and how you are doing it. Sometimes just having to explain the code line-by-line will reveal anomalies that you had not previously seen. Even doing this on your own can be a valuable exercise.

9.5 DOCUMENTATION

Following programming, standards go hand in hand with documenting the application. They are equally important. Documentation lets users know how to use the application and helps developers and support personnel know how to maintain it. Four levels should be developed: technical, code, user, and end-user help. We look at each one in turn.

9.6 TECHNICAL

Technical documentation details how the application is laid out, how it works, and what each piece of the program does along with its relationship to other parts of the project. In large part, the documentation you developed in chapter 6 will provide the basis for this type of documentation. Other useful elements include:

- a list of all files used in an application and their functions
- a road map of how the files interrelate
- annotated source code for all scripts
- fully documented function libraries, where appropriate
- testing documents and procedures.

9.7 CODE

Your program code must be documented to help programmers. Programmers, being the creative people that they are, would far rather code something new than document code they've already written. This natural tendency needs to be overcome if you want supportable applications. Having a code review process in place can help provide the discipline needed to make this happen. If developers comment on the code as they go, the going is that much easier. Initially, preliminary comments at given points are generally adequate. As sections are completed, however (perhaps preparatory to a code review), developers should write concise yet complete comments. If changes are made to an application, existing comments in the code should be reexamined to make sure that they are still accurate.

9.8 USER

Particularly in more complex applications, users need to have support materials to help them to know what the application does, what the input fields mean and what they do, how to navigate the application, and how to know where certain tasks are accomplished. Having such documents available is critical to user training, particularly for staff that may not be familiar with the application.

One methodology is to take the roadmap created in the technical documentation above and proceed through the application, taking screenshots of each step. Then, for each screenshot, describing each field, indicating what it does and what alternatives it provides. You can also indicate where alternate tasks are supported and where multiple branches can be taken, referring to the relevant screenshot at the appropriate point.

9.9 HELP SCREENS

Pairing written and online documentation is also useful. Given the hypertext basis of the Web, it is quite easy to provide context-sensitive help at any point in the program with hyperlinks embedded at appropriate places in the application pages.

9.10 DEBUGGING

No matter how good a programmer you are, you won't always get it right the first time. Things may not work as you expect; values don't get added to the database or, if they are, they are incorrect. Several techniques that you can use with PHP and MySQL can help you track such problems down quickly. The most useful approach to debugging is to start at the beginning of the script and work your way through. The essential technique is to run a few lines of code, use `echo` to print out how things are doing, and immediately use the `exit` to stop script processing. If everything is okay, you move the `echo/exit` pair down a little in the script to check things out. The moment you encounter something you don't expect, you know that the problem is between the place where you last got what you expected and the position where it failed. You want to be sure to check for several items:

- Is a loop being entered? Placing an `echo "HERE I AM"` may not be exciting, but it does at least tell you if your loop is being entered. If it isn't, you need to check your conditions to make sure things are getting set as they should be, and that you don't have a typo or have misnamed a variable.
- Are you not getting what you expect from an SQL query? Try either echoing it to the screen or using the `mysql_num_rows ()` function to learn how many records are being returned by the query. If the number is 0, there aren't any matching records, meaning that you either need to add records or there is a mistake in your query.
- Is an error message not appearing in the browser? Some times MySQL simply won't do this. One workaround technique I have found useful is to `echo` the

\$query to the screen and then cut and paste it into phpMyAdmin's SQL box, and then run it there.

- Has a variable been set? Echoing a variable can also let you know whether it has been set. Using printable characters on either side of the variable is sometimes useful. For example, using the line `echo "$var";` will print out `**` if the variable is empty.
- Are queries querying? One technique I have used throughout the examples in this book (and have included in the function library) has been to use `or die` (the `mysql_error()` function) each time I run a query. Doing so outputs the error the script gets from MySQL, prints it out on the screen, and then exits. This can be an extremely useful debugging technique.

9.11 SELF-ASSESSMENT QUESTIONS

- Q.1 Describe the program development procedure, its establishing process and implementing standards with examples.
- Q.2 Explain the key factors of documenting the programming development procedure.
- Q.3 Define the following:
- Establishing a process
 - Implementing standards
 - Quality assurance
 - Documentation (technical, code, users, help screens and debugging).

9.12 ACTIVITY

Enlist important practices while you're writing the code for your library application.

9.13 REFERENCES

Baruah, A. (2002). *Library database management*. Gyan Publishing House.

Krier, L., & Strasser, C. A. (2014). *Data management for libraries: LITA guide*. Chicago: American Library Association. Available at:
<https://books.google.com.pk/?hl=en>

Preston, C., & Lin, B. (2002). Database technology in digital libraries. *Information Services & Use*, 22(1), 9-17.

Singh, P. (2004). Library databases: Development and management. *Annals of Library and Information Studies*, 51(I), 72-81. Available at:
<http://nopr.niscair.res.in/bitstream/123456789/7488/1/ALIS%2051%282%29%2072-81.pdf>

Suseela, V. J., & Uma, V. (2017). *Data management for libraries: Understanding DBMS, RDBMS, IR technologies & tools*. Ess Ess Publications.

Westman, S. R. (2006). *Creating database-backed library web pages: Using open source tools*. Chicago: American Library Association. Available at:
[https://pdfdrive.com/download.php?title=Creating%20DatabaseBacked%20Library%20Web%20Pages:%20Using%20Open%20Source%20Tools%20\[1ed.\]0838909108,%209780838909102,%209780838998489&content=&downlurl=](https://pdfdrive.com/download.php?title=Creating%20DatabaseBacked%20Library%20Web%20Pages:%20Using%20Open%20Source%20Tools%20[1ed.]0838909108,%209780838909102,%209780838998489&content=&downlurl=)